Classification and Regression

From linear and logistic regression to neural networks

Project 2 - FYS-SKT4155

Paulina Tedesco and Hanna Svennevik

November 2018

Contents

List of Figures

List of Tables

Chapter 1

Abstract

This study provides an assessment of various state of the art methods used in machine learning for solving regression and classification problems, applied to the widely used Ising model. First, we estimate the coupling constant of the one-dimensional Ising model by applying linear regression and neuronal networks. The LASSO is the only approach that reproduces the original pattern from which the data was generated, breaking the symmetry. The optimal regularisation parameter is = 0.001, resulting in an $MSE = 3.07$, much lower than the error of the other two schemes (the OLS and ridge). Neuronal networks improve slightly the results giving, in the best case, an $MSE = 2.48$- without regularisation, using $\eta = 0.001$, 50 epochs, and a batch size of 10 data points.

In the second part, we determine the phase of the two-dimensional Ising model in a $40x40$ lattice. Logistic regression and neuronal networks are used for this purpose. For 50 epochs, ridge performs better than LASSO (with an accuracy of 0.70 versus 0.68). However, the highest accuracy achieved with logistic regression is 0.71 (LASSO, 10 epochs). On the other hand, MLPClassier gave an accuracy of $0.\overline{99}$ for the best subset of hyper-parameters. This is nearly a perfect classifier.

In both cases, we explore different ranges of the hyperparameters, such as the learning rate, the number of iterations, and the regularisation parameter, attempting to find the optimal values for each problem. A variety of activation functions were tested, but we present our main results for the sigmoid function.

The research accomplished by Mehta et al. (2018)[9], validates our results and, consequently, the implementation of the algorithms, which is a central part of this work. We emphasise that a big part of the value of this study relies on the notebooks and the developed code, available on the GitHub repository for further contributions of the scientific community, as well as comparisons using different sets of hyperparameters and/or potentially new methods.

Chapter 2

Introduction

Machine learning is one of the fields of science that is developing fastest in the last years, and at the same time, the recognition of its value increases. A vast variety of applications is surging due to the advances in machine learning, not only in other research areas, but also in the financial industry, health care, public safety, transportation, and marketing, just for mentioning a few examples. Our aim is to give an overview of the algorithms used in supervised learning, in particular regression and classification problems, using the Ising model, which has originally a motivation in ferromagnetism, but later was reinterpreted in many different ways[17].

It is also of our interest to develop the code for different regression and classification techniques using Python, which "has a solid claim to being the fastest-growing major programming language"[14]. The graph from 2017 in figure 2.1 illustrates the relatively rapid increase in the use of Python compared to five of the ten more popular programming languages, as well as future projections along with their 80% prediction interval. The visits to questions in Stack Overflow was used as a proxy, rather than questions asked (which has more month-by-month noise). We can see that Python passes from being the less visited tag in 2012 on Stack Overflow to the most visited in June of 2017 within high-income countries.

In 2018, according to the Developer Survey Results carried out by Stack Overflow[15], the most loved and less dreaded framework was TensorFlow (see figure 2.3). This conclusion was obtained by investigating what proportion of developers that currently work with a technology do or do not want to continue to do so. TensorFlow is a relatively new machine learning library, released as open source by Google in 2015, and has already become popular among data scientists. It is typically used for deep learning, and the reader can easily compare our results to those obtained with TensorFlow, and continue building upon that. Exhibiting one of the highest year-over-year growth rates ever in questions asked on Stack Overflow, it is a demonstration of the rise of tools for machine learning.

Figure [15] shows the correlation between technologies and the use of TensorFlow. The most highly correlated technology is Torch/PyTorch; which is a competing framework developed and released by Facebook. Next com Jupyter and Python, also used in this study.

We compare all the results with the obtained by Mehta et al. (2018)[9], and also to well established python libraries. Our code, as well as some notebooks with examples, are provided in the Github repository:
https://github.com/hannasv/project2

Three common approaches were used in the regression analysis: the OLS, ridge and the LASSO; while the classification was performed using logistic regression. A neuronal network was also set up for each of the problems using one single hidden layer. We present our main results for the sigmoid function, but other activation functions were also tested. The stochastic gradient descent method with minibatches was preferred, although the steepest descent was also implemented. The results are discussed in terms of the hyperparameters.

This document is structured as follows. The data and the algorithms are presented in the section methods. At the end of that section, the python implementation for the analysis is described. The result section consists of the most interesting outcomes of the analysis and is divided into two sections, one for the regression and another for the classification analysis. The project is summarised in the conclusion section, were we also state some future analysis that we would like to work on.

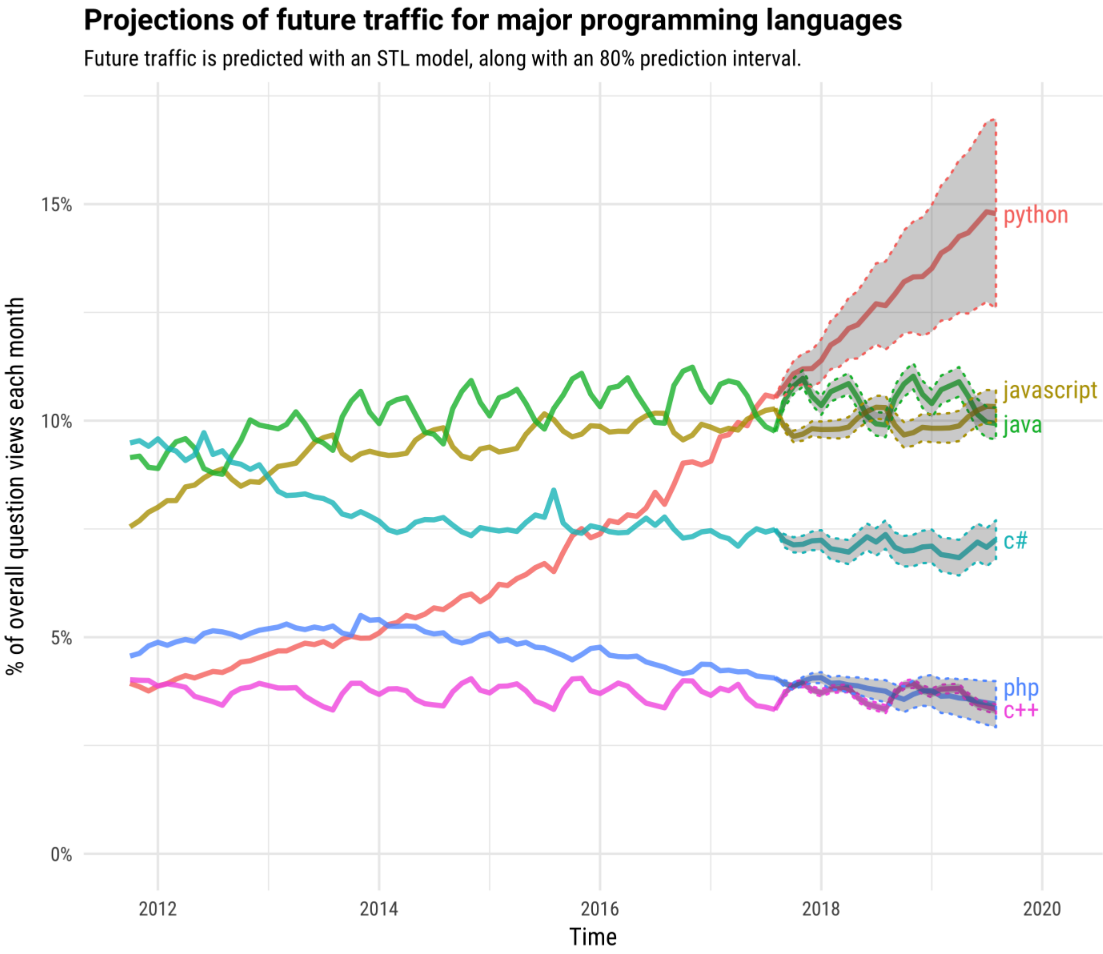

Figure 2.1: Projections of future traffic for major programming languages usingan STL model, along with an 80% prediction interval. It is based on Stack Oveflow question views in World Bank high-income countries[14].

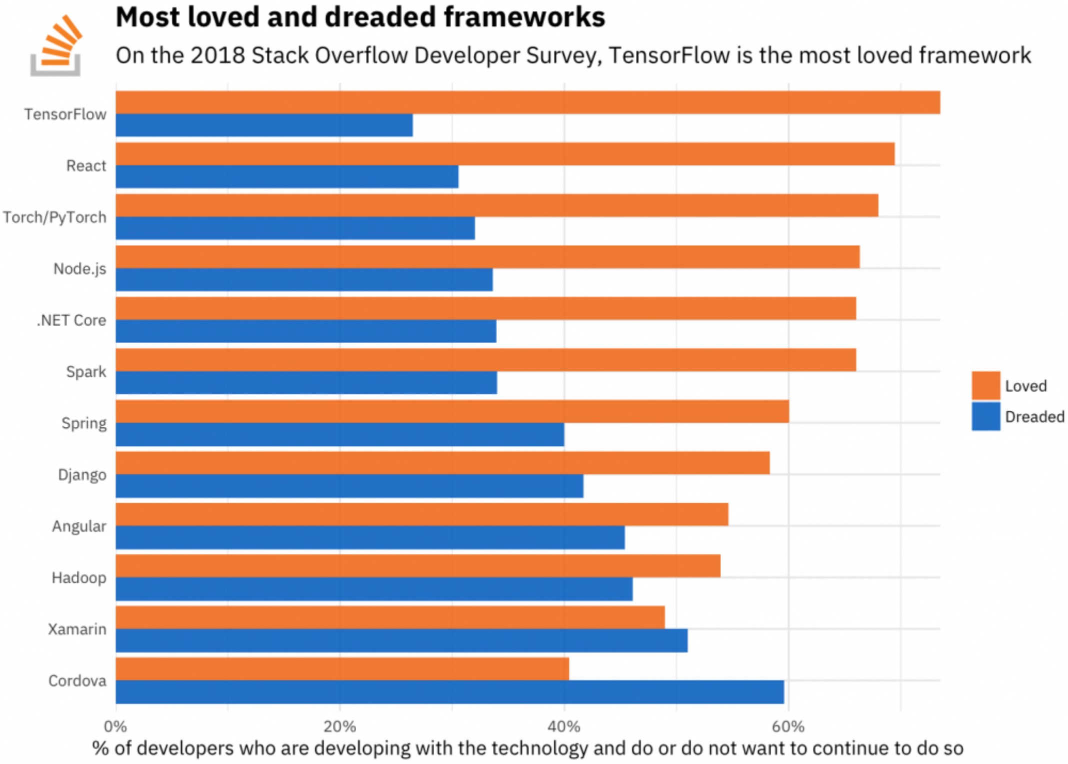

Figure 2.2: Most loved and dreaded frameworks on 2018 Stack Overflow Survey[15].

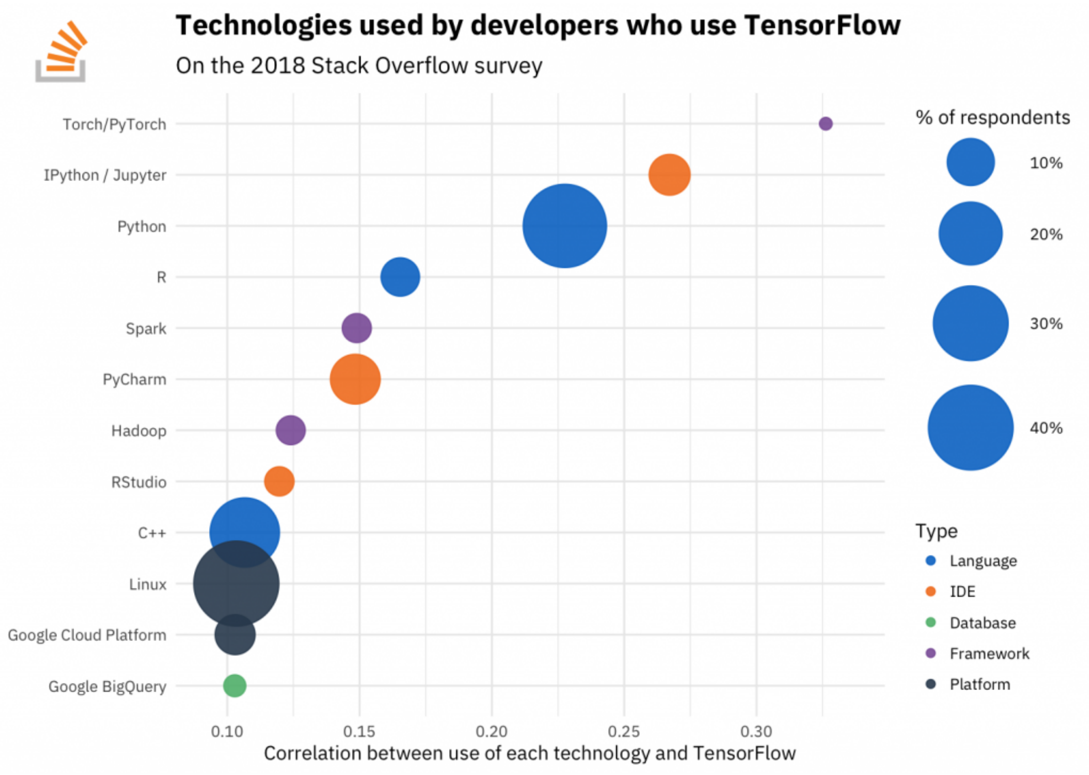

Figure 2.3: Correlations between the technologies used by developers who use TensorFlow and this library[15].

Chapter 3

Methods

3.1 Data

Our research is based on the Ising model, named after the physicist Ernst Ising. The variables of this model, which in physics often are thought of as magnetic dipole moments of atomic spins, can take two possible values in a binary model, either 1 or -1. The spins, arranged in a lattice, are only allowed to interact with their neighbours.

The one-dimensional Ising model (solved by Ising in 1925) is applied in the first part of this study, the regression analysis, whereas the classification part is performed using the two-dimensional model (the analytic description was provided by Lars Onsager in 1944). All thermodynamic functions of the model can be calculated from this expression for the free energy. An interesting fact of this model is that it was the first to exhibit a continuous phase transition at a positive temperature, T_c.

Two simplifications are usually made. The first one is that no external field interacts with the lattice, for all the sites, j; as a consequence, the Ising model is symmetric under switching the value of the spin in all the lattice sites. The second simplification consists of assuming that all the neighbours, $< ij >$ have the same interaction strength, then the interaction $J_ij = J$ for all the pairs ij. J is arbitrarily set to 1. [17]

3.1.1 The one-dimensional Ising model

The Hamiltonian for the one-dimension Ising model with nearest-neighbour interactions on a chain of length L, periodic boundary conditions $S_j \in \pm 1$ and Ising spin variables is given by:

$$E[\hat{s}] = -J \sum_{j=1}^{N} s_j s_{j+1}, \quad S_j \in \pm 1 \tag{3.1}$$

where the lattice indices are i, j, and J is an arbitrary energy interaction scale. We assume that all sites have the same number of neighbours due to periodic boundary conditions, which are often chosen for approximating a large (infinite) system by using a small part called a unit cell.

After setting $J = 1$, a large number of spin configurations are drawn, and their Ising energy is computed. Assuming that the Hamiltonian is unknown, our goal is to learn a model that predicts E_j from the spin configurations with linear regression, given a set of $i = 1, ..., n$ points of the form $\{H[S^i], S^i\}$.

Before learning the model, it is necessary to choose the model class. One intuitive and sensible choice would be to look at a spin model with pairwise interactions between every pair of variables.

$$H_{model}[S_i] = -\sum_{j=1}^{L}\sum_{k=1}^{L} J_j, kS_j^iS_k^i \tag{3.2}$$

The problem is well defined since the unknown $J_{j,k}$ that we want to model enters linearly in the Hamiltonian. We can, therefore, use linear regression analysis, but we need to recast the model in the form:

$$H_{model}[S_i] = X^i J, \tag{3.3}$$

where i runs over the samples in the dataset and the vectors X^i represent all two-body interactions $\{S_j^iS_k^i\}_{j,k=1}^L$. Note that the dot product can be represented by a single index $p = j, k$. [17]

3.1.2 The two-dimensional Ising model

For the classification problem, we have the two-dimensional Ising model, given by 3.4.

$$E = -J\sum_{<ij>}^{N} s_is_j, \quad S_j \in \pm 1, \tag{3.4}$$

In this equation the lattice site indices i, j run over all nearest neighbours of a 2D square lattice with periodic boundary conditions. As in the one-dimensional model, the energy scale J is arbitrarily chosen.

We use the same data set as in Mehta et al. (2018) [9], which consists of 10^4 states at fixed temperatures, T, generated using the Monte Carlo method on a lattice of dimensions $40x40$. This data is available for all temperatures in the range from $T = [0.25, 4]$ with a step of 0.25 units of energy.

There are three possible phases for the spin configuration depending on the temperature: ordered, with all the spins aligned; disordered; and a thermal transition phase at a critical temperature T_c, discovered by Onsager.

$$T_c/J = 2/log(1+\sqrt{2}) \approx 2.26, \tag{3.5}$$

This critical point is extended to a critical region around T_c, for any finite system size. The following three states are considered: ordered ($T_c/J < 2.0$), near-critical ($2 < T_c/J < 2.5$), and disordered ($T_c/J > 2.5$). The labels 0 and 1 are assigned to the states disordered and ordered respectively. For simplicity, temperatures leading to a critical phase will be avoided in this analysis. Previous studies state that it is expected to be harder to identify the phase in this region.

Given a certain spin configuration from the two-dimensional Ising model, our goal in the second part of this study is to classify the corresponding phases (i.e., ordered or disordered), using logistic regression and a multilayer perceptron classifier. This analysis is performed on a flattened 1D array of the two-dimensional state, even though it does not allow us to learn the structure of the contiguous ordered 2D domain. [17]

3.1.3 Objective functions

Machine learning problems often involve optimising an objective function, C. The objective function is a mathematical expression that describes the problem we want to optimise. Since classification and regression involve very different tasks, they are solved by using different objective functions. The objective of a classification problem is to maximise the likely-hood of being in a specific class, whereas in regression problems we minimise the squared distance between the data and the fit. The latter is also known as the residual sum of squares, RSS. Both cases allow for the addition of a penalty, λ. The purpose of the penalty is to get a good fit while the weights are retained to low values.

3.2 Regression analysis

Ordinary least squares

In the Ordinary Least Squares (OLS), $\hat{Y}$ denotes the predicted model given the input $\mathbf{X}$, and $\hat{\beta}_0$ is the intercept also known as the bias in machine learning. The best fit is given by the $\hat{\beta}$-values that minimise the residual sum of squares, which is the cost function in this case. The minimum is found by differentiating the RSS with regard to β and setting it equal to zero. Finally, solving for $\hat{\beta}$ results in equation 3.8. This has a unique solution when $\left(\mathbf{X}^T\mathbf{X}\right)$ is non-singular. A matrix is non-singular when X^TX has full rank, then all the columns are linearly independent and the matrix is invertibleHastie, Tibshirani, and Friedman (2001) [4].

$$\hat{Y} = \hat{\beta}_0 + \sum_{j=1}^{p} X_j \hat{\beta}_j \tag{3.6}$$

$$RSS(\hat{\beta}) = (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta) \tag{3.7}$$

$$\hat{\beta} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T\mathbf{y} \tag{3.8}$$

Ridge regression

When the amount of predictors is large relative to the number of observations, a penalisation term is often added to penalise those features that add little to the prediction, by adding a penalty term, λ, to the diagonal of the matrix. The ridge regression imposes a size constraint on the coefficients and reduces the number of correlated coefficients which could result in a singular matrix. Hastie, Tibshirani, and Friedman (2001)

$$RSS(\hat{\beta}) = (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta) + \lambda\beta^T\beta \tag{3.9}$$

$$\hat{\beta}^{ridge} = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1} \mathbf{X}^T\mathbf{y} \tag{3.10}$$

The LASSO regression

The difference between the LASSO and ridge regression is that the regularisation term in the cost function is in absolute value. The $\sum_{j=0}^{p} \beta_j^2$ is replaced with $\sum_{j=0}^{p} |\beta_j|$. Lasso is non-linear and there is no closed expression for $\hat{\beta}$ in this case. [4] The expression for Lasso on Lagrangian form is:

$$\hat{\beta}^{lasso} = \underset{argmin}{\beta} \left\{ \frac{1}{2} \sum_{i=1} \left(y_i + \beta_0 + \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=0}^{p} |\beta_j| \right\} \tag{3.11}$$

Another substantial difference between ridge and the LASSO is that the former shrinks the coefficients, but not to zero, whereas the latter does not only penalise the coefficients but actually set them to zero if they are irrelevant. Hastie, Tibshirani, and Friedman (2001)

Performance metrics

In order to quantify the performance of a regression model we use metrics. In this article, we use both the mean square error, MSE, and the R^2-score. MSE is the average squared distance between the predicted and the true value. The R^2-score is a measure on how well future samples are likely to be predicted by the model. They both result in a good score when the models' performance approaches the true value. Then the MSE goes to zero and the R^2 goes to one. The metrics can be calculated by the following equations. [5]

$$MSE(\hat{y}, \hat{\tilde{y}}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 \tag{3.12}$$

$$R^2(\hat{y}, \tilde{\hat{y}}) = 1 - \frac{\sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2}{\sum_{i=0}^{n-1} (y_i - \bar{y})^2} \tag{3.13}$$

where mean value of $\hat{y}$ is defined as $\bar{y} = \frac{1}{n} \sum_{i=0}^{n-1} y_i$.

3.3 Singular Value Decomposition

The Singular Value Decomposition (SVD) is a numerical method that allows us to construct a factorisation of a given matrix X. The expression of the factorisation is given in equation 3.14. For a non-singular square matrix X, the matrix $X^T X$ is symmetric and positive definite. Thus, the eigenvalues of $X^T X$ are positive real numbers. The singular values can be calculated by taking the square root of the eigenvalues. The matrix Σ is diagonal, and its elements are the singular values ordered in decreasing order. Here, the matrix V consists of the eigenvectors, whereas the matrix U can be calculated by $U = AV\Sigma^{-1}$. Both U and V are rectangular matrices consisting of orthonormal columns. Since its more desirable to have square matrices, we extend U and V with a vector which is orthonormal to the other columns. The singular value decomposition is not unique, because of several options for the eigenvalues.

$$X = U\Sigma V^T \tag{3.14}$$

Calculating the eigenvalues of $X^T X$ implies a large computational cost. Other problems may arise from loss of accuracy, $X^T X$ might be singular and, lastly, a sparse X can result in a dense $X^T X$. Faul (2016), [1].

3.3.1 Psudoinverse of SVD

We often run into problems since the inverse of a singular does not exist. One solution to this problem is to use the Moore-Penrose pseudoinverse. The Moore-Penrose pseudoinverse for a singular value decomposition is defined in equation 3.15.

$$X^+ = \frac{X^T}{X^T X} = \frac{\left(U\Sigma V^T\right)^T}{\left(U\Sigma V^T\right)^T \left(U\Sigma V^T\right)} = \frac{V\Sigma U^T}{\Sigma^2} = V\Sigma^+ U^T \tag{3.15}$$

The Moore-Penrose pseudoinverse exists and is unique for both square and rectangular matrices. Golan (2012) [3].

Ordinary least square

From section 3.2 we know that in order to solve the analytical expression for the regression coefficients we need to invert a matrix. In most cases, this is not possible. By using the pseudoinverse we can expand the set of problems we can find a solution to. Using the pseudoinverse to calculate the regression coefficients we obtain the expression in . [8]

$$\omega = X^+ y = \frac{V\Sigma U^T}{\Sigma^2} y = V\Sigma^+ U^T y \tag{3.16}$$

Ridge

It is possible to obtain a similar expression for the regression coefficients when using ridge regression as well. This is shown in equation 3.17. [8]

$$\omega = \frac{X^T}{X^TX+\lambda}y = \frac{V\Sigma U^Ty}{\left(V\Sigma U^T\right)^T V\Sigma U^T+\lambda} = \frac{V\Sigma U^T}{\Sigma^2+\lambda}y = V\Sigma U^Ty\left(\Sigma^2+\lambda\right)^{-1} \tag{3.17}$$

3.4 Logistic regression

Logistic regression is a method of classification, not regression method as its name suggests. In this article, we want to use logistic regression to determine the state of spin configurations in the Ising model. As we discussed in section 3.1.2, we restrict ourselves to the cases: ordered and disordered phase. The outcomes are labeled according to equation 3.18. This can be used to solve linear and binary classification problems. In other terms, we assume that the problem has two outcomes which can be separated with a linear decision boundary.

$$y_i = \begin{pmatrix} 0 & disordered \\ 1 & ordered \end{pmatrix} \tag{3.18}$$

The sigmoid function is an example of a soft classifier, and it returns the probability of being in a class. An important property of the sigmoid function is the probability $p(-t) = 1 - p(t)$. Here, t denotes the linear combination between weights and sample features. The expression for the sigmoid function is displayed in equation 3.19.

$$p\left(t\right) = \frac{1}{1+e^{-t}} = \frac{e^t}{1+e^t} \tag{3.19}$$

In this article we will build a model that takes spin configurations as input and predicts whether the spin configuration belongs or not to an ordered or a disordered phase. Expanding the classification problem to include p predictors results in the following probability functions.

$$p\left(y_i = 1|x_i, \hat{\omega}\right) = \frac{e^{\omega_0+\omega_1x_i+\cdots+\omega_px_i}}{1+e^{\omega_0+\omega_1x_i+\cdots+\omega_px_i}} \tag{3.20}$$

$$p\left(y_i = 0|x_i, \hat{\omega}\right) = 1 - p\left(y_i = 1|x_i, \hat{\omega}\right) = \frac{e^{-\omega_0-\omega_1x_i-\cdots-\omega_px_i}}{1+e^{-\beta_0-\omega_1x_i-\cdots-\omega_px_i}} \tag{3.21}$$

As mentioned in section 3.1.3, in logistic regression we want to maximise the likely-hood of our model to perform the correct classification. The cost function is expressed as the log-likelihood of a specific event.

$$C_{ols}\left(X,\omega\right) = \sum_{i=1}^{n}\{-y_ilog\left(f\left(x_i^T\omega\right)\right) - \left(1-y_i\right)log\left[1-f\left(x_i^T\omega\right)\right]\} \tag{3.22}$$

Introducing a penalty makes it possible to control how we fit training data and still keep the weights small. Adding ridge and lasso penalties results in the following expression.

$$C_{ridge}\left(X,\omega\right) = \sum_{i=1}^{n}\{-y_ilog\left(f\left(x_i^T\omega\right)\right) - \left(1-y_i\right)log\left[1-f\left(x_i^T\omega\right)\right]\} + \frac{\lambda}{2}\sum_{j=1}^{n}\omega_j^2 \tag{3.23}$$

$$C_{lasso}\left(X,\omega\right) = \sum_{i=1}^{n}\{-y_ilog\left(f\left(x_i^T\omega\right)\right) - \left(1-y_i\right)log\left[1-f\left(x_i^T\omega\right)\right]\} + \lambda\sum_{j=1}^{n}|\omega| \tag{3.24}$$

Taking all the partial derivatives of the cost function, C, with respect to all the predictors ω. Here $\hat{p} = f(X\omega)$.

$$\frac{\partial C_{ols}\left(\hat{\omega}\right)}{\partial\hat{\omega}} = -\hat{X}^T\left(\hat{y}-\hat{p}\right) \tag{3.25}$$

$$\frac{\partial C_{ridge}\left(\hat{\omega}\right)}{\partial\hat{\omega}} = -\hat{X}^T\left(\hat{y}-\hat{p}\right) + \lambda\omega \tag{3.26}$$

$$\frac{\partial C_{lasso}\left(\hat{\omega}\right)}{\partial\hat{\omega}} = -\hat{X}^T\left(\hat{y}-\hat{p}\right) + \lambda sgn(w) \tag{3.27}$$

In order to perform the classification we use stochastic gradient descent and update the weights based on the gradient of the cost function with respect to the weights. You can read more about this in section 3.4.2

It is a necessary condition to have a convex function in order to use the stochastic gradient descent to solve the optimisation problem. To determine if the cost function is convex on a can take a look at the sign of the double derivative of the cost function with respect to the weights.

$$\frac{\partial^2 C_{ols}\left(\hat{\omega}\right)}{\partial\hat{\omega}\partial\hat{\omega}^T} = \hat{X}^T W \hat{X} \tag{3.28}$$

where W is a diagonal matrix with elements $p\left(y_i|x_i,\hat{\omega}\right)\left(1-p\left(y_i|x_i,\hat{\omega}\right)\right)$. [7]

3.4.1 Accuracy

The performance of a classifier can be evaluated by applying the accuracy score on the test data. The term classifier includes both logistic regression and multilayer perceptron. This score tells us the proportion of correctly classified data with respect to the total. Hence, a perfect classifier will have an accuracy score of 1 [6]. The accuracy can be calculated by equation 3.29, where t denotes the target values and y the predicted values.

$$Accuracy = \frac{1}{N}\sum_{i=1}^{N} I(t_i = y_i) \tag{3.29}$$

3.4.2 Gradient descent

It is not always possible to minimise the cost function analytically, this means that we must apply some numerical method to find the minimum of the function. The easiest method to implement is the Newton-Raphson, but it requires an efficient computation of the Jacobian and the Hessian matrices. However, this iterative method is not recommended when the derivatives only can be calculated numerically and/or the function is not smooth. If the first guess is close enough to the solution, the Newton-Raphson method converges rapidly, otherwise, the method can lead to inaccurate results. Furthermore, when the guess is far from the solution, but near a local extreme, the method will fail.

If the cost function is convex, we can use an optimisation algorithm called gradient descent to find the parameters that minimises the cost functions in the classification problem[13], obtaining low loss and high classification accuracy. It consists of taking a step in the opposite direction of the gradient in each iteration, and update the weights. The learning rate, η, and the slope of the gradient will determine the step size (see figure **??**). The weight change is defined as the negative gradient of the cost function multiplied by the learning rate. In this study, we maintained a fixed ηover all the iterations and updated the weights simultaneously.

Given the function that we want to minimise, $E(\theta)$, in our case the cost function, the algorithm consists of initialising the parameters to θ_0 and thereafter update them according to the following expression [9]:

$$\begin{aligned} v_t &= \eta_t \nabla_\theta E(\theta_t) \\ \theta_{t+1} &= \theta_t - v_t \end{aligned} \tag{3.30}$$

To find the optimal learning rate, we run the same code for different values of η and then compare the results. On the one hand, if we choose a too large learning rate, the error becomes larger in every epoch (number of iterations) because we overshoot the global minimum (see illustration on the right panel of figure **??**) and panel d in figure 3.1. On the other hand, if the learning rate is too small, the algorithm would require a very large number of epochs to converge to the global minimum[13]. If we are lucky and hit the optimal learning rate, it will converge in one singles step. A learning larger than the optimal but smaller than two times the optimal will oscillate around the minimum and eventually converge (see panel c of figure 3.1.

There are several variants of the gradient descent method. In this section, we discuss the methods implemented for our study, the standard gradient descent and the stochastic gradient descent with mini-batches. All

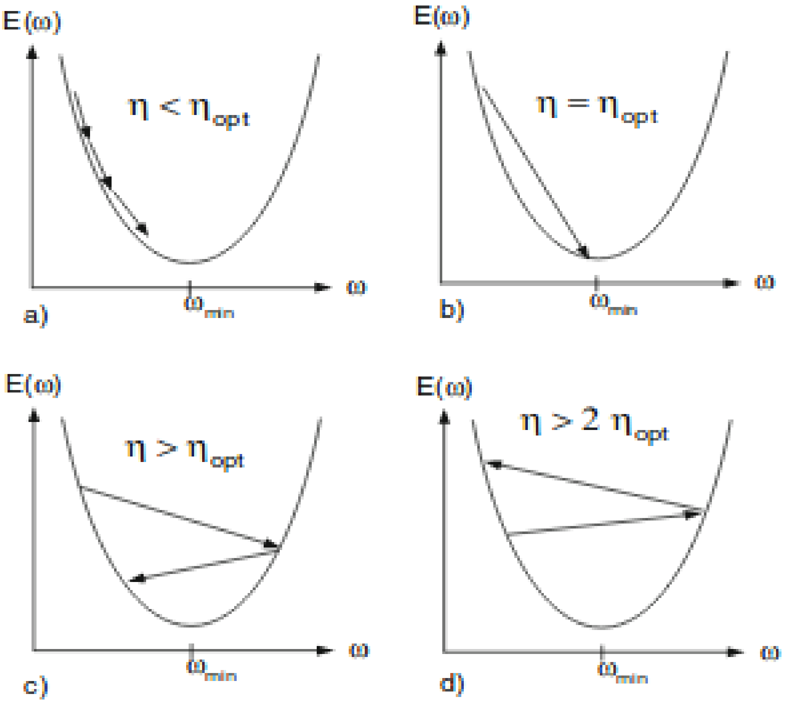

Figure 3.1: Illustrations of the different scenarios of gradient descent based on the different learning rates. Montavon, Orr, and Müller (2012)[9]

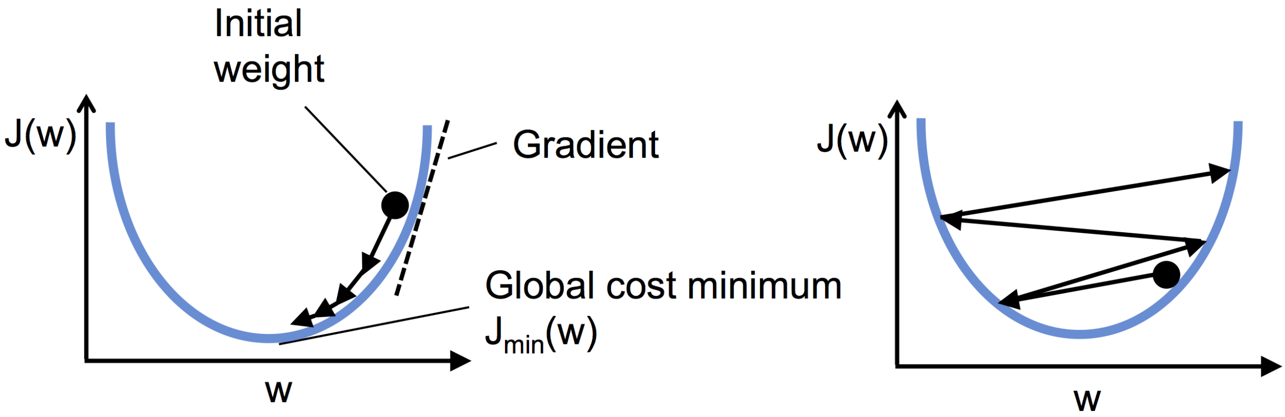

Figure 3.2: Illustration of the convergence of the gradient descent method (in the left panel) for a small enough learning rate, and the non convergence when the learning rate is too large (right panel)[13]. In this context, J is the cost function and w the weights, as usual.

the methods have in common that the goal is to try different values of the weights, evaluate their loss, and in the next iteration update the weights as explained before, in order to decrease their loss [12]. The steps are repeated until the parameters reach the minimum loss and high classification accuracy, First, some useful notation of the parameters used in our function:

- loss : A function used to compute the loss over our current parameters W and input data
- data : Our training data where each training sample is represented by a feature vector.
- w : This is actually our weight matrix that we are optimising over. Our goal is to apply gradient descent to find a w that yields minimal loss.

Our code of gradient descent methods iterates until the initialised number of epochs is reached. This can be optimised by introducing a condition with, for example, the maximum number of iterations without movement. The gradient is evaluated given the parameters listed above, then the weights are updated by subtracting the gradient multiplied by the learning rate. This means that the learning rate controls the step size. The details are explained in the subsection Implementation.

The steepest gradient descent method is not suitable for large data sets because it calculates the prediction of each point in the training set. A more efficient method implemented in this work is the stochastic gradient descent. The term stochastic alludes to the shuffle of the data in each iteration. This causes some noise in the updates of the weights, but since we loop over small batches instead of the entire data set, we can also take more steps along the gradient. Thus, leading to faster convergence without any negative effects on the loss and accuracy.

3.5 Neural networks

3.5.1 The perceptron rule

Neurons are interconnected nerve cells in the brain that transmit chemical and electrical signals. The first model of a simplified brain cell was published in 1943 by Warren McCulloch and Walter Pitts[13]. They described the neurons as a logic gate with a binary output; the signals reach the dendrites, and are transmitted to the axon; only if the signals exceeds a certain threshold, it will generate and pass an output by the axon. See the model of a neuron in the following figure 3.3.

A perceptron rule based on the previous model was published in 1957 by Frank Rosenblatt[13]. The algorithm

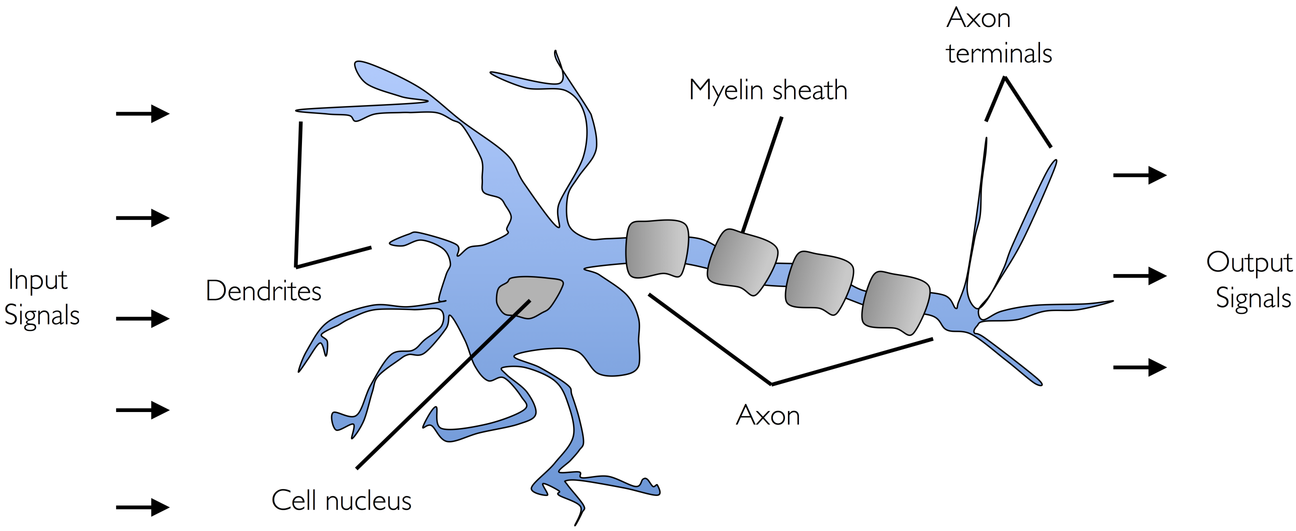

Figure 3.3: Model of a neuron [13].

he proposed learns automatically the optimal weight coefficients involved in the decision of whether the neuron fires or not. This algorithm can be used to predict to which class the sample belongs. A schematic representation of the perceptron for three inputs x_1, x_2, x_3 is shown in figure [11], but it can have more or fewer outputs.

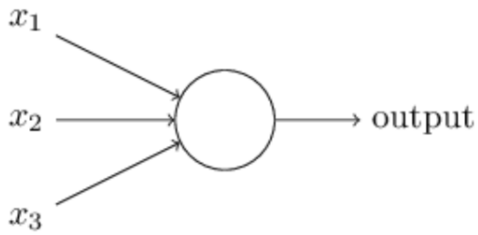

Figure 3.4: Perceptron model [11].

More formally, this model can be written as follows:

$$y = f(\sum_{i=1}^{n} w_i x_i) = f(u) \tag{3.31}$$

where the output, y, is the value of the activation function applied to the weighted sum of the signals $x_i, i = 1 : n$ received from other neurons, which can be written as, $z = w_1x_1 + w_2x_2 + ... + w_nx_n$. [6] If the net input of a sample $x^{(i)}$ is greater than the defined threshold θ , we predict class 1, and class 0 otherwise:

$$y = \begin{cases} 0, & \text{if } \sum_{i=1}^{n} w_i x_i \leq \theta \\ 1, & \text{if } \sum_{i=1}^{n} w_i x_i > \theta \end{cases}$$

The system above can be simplified by writing the sum as a dot product of vectors $W.X$, and moving the threshold to the other side of the inequality (known as the bias, b).

$$y = \begin{cases} 0, & \text{if } X.W + b \leq 0 \\ 1, & \text{if } X.W + b > 0 \end{cases}$$

Figure 3.5 shows, in the left panel, the binary output, 1 or 0, after applying the activation function (represented in the figure with the letter ϕ) to the weighted sum of signals; the right panel shows how the model can be used to discriminate between two linearly separable classes.

The output value is the class label predicted by the step function defined above. In the algorithm, the simultaneous update of the weights can be written as:

$$w_i := w_i + \Delta w_i \tag{3.32}$$

And Δw_i is calculated by the perceptron learning rule:

$$\Delta w_i = \eta(y^{(j)} - \hat{y}^{(j)})x_j^{(i)} \tag{3.33}$$

Here, η is the learning rate, $y^{(j)}$ is the true class label of the training sample j, and $\hat{y}^{(j)}x_j^{(i)}$ is the predicted class label.

The general idea of the models is to mimic how the neurons in the brain behave when it receives many signals from the neurons it is connected to, this is, it fires or not. The convergence of the perceptron is only guaranteed if the classes are linearly separable and the learning rate is small enough. The following diagram summarises the perceptron model. It illustrates how the net input is calculated from the combination of the inputs of the sample x with the weights w. The net input is passed to the threshold function in order to generate the binary output 0 or 1. Then, the prediction error is calculated from the output and the weights are updated during the learning phase.

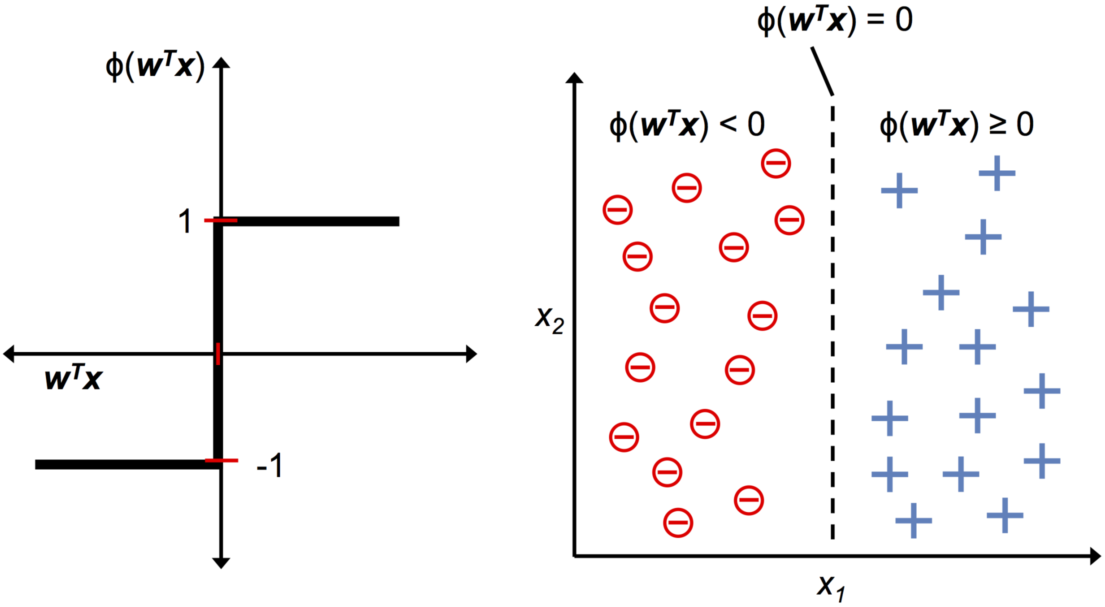

Figure 3.5: Representation of the binary output after applying the decision function (left panel), and how it is used to discriminate classes (right panel) [13].

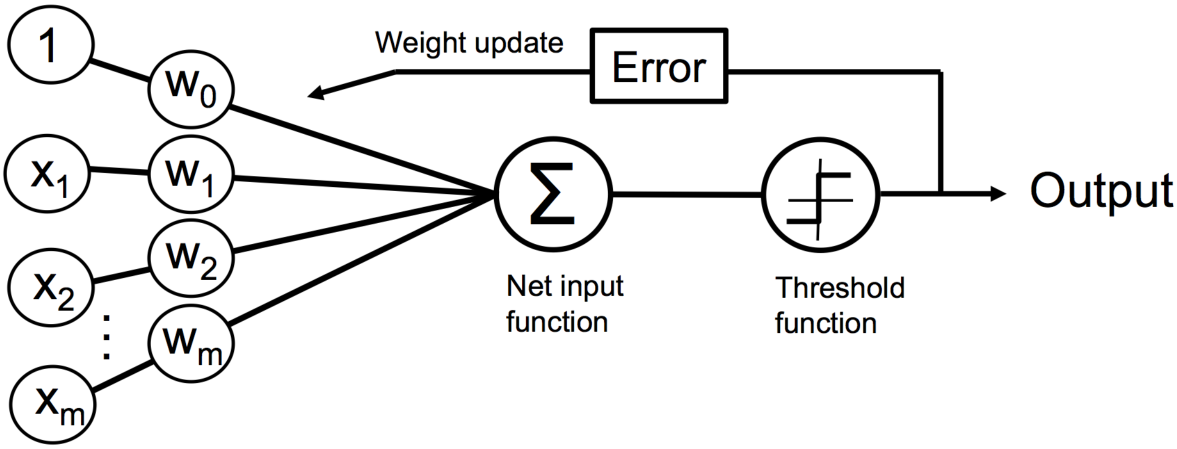

Figure 3.6: Diagram of the perceptron model [13]. The threshold function receives the net input, and generates a binary output. Then the error is calculated and the weights are updated.

3.5.2 Sigmoid and other activation functions

The concept of sigmoid neurons is similar to the perceptron model represented in figure 3.4, and it is introduced to make small changes in bias and weights cause only small changes in the output (this is not always the case of the perceptron model). The difference between the perceptron and the sigmoid neurons is that for the latter the output is not 0 or 1 but $\sigma(W.X + b)$, where σ is the sigmoid function defined by[11]:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \tag{3.34}$$

The shape of this function is a smoothed out version of the step function (see figure 3.7). As a consequence, small changes in the output for small changes in the weights and bias. It is the shape of the function, and not its exact form, which is important. Thus, there are several activation functions that can be applied in neuronal networks (see figure 3.7). However, the algebra is simpler when the sigmoid function is used because of the properties of the exponential, and it is widely used in the literature.

The layers of deep neuronal networks may learn at different speeds. It might happen that the gradients get smaller for each iteration and that the training process does not converge to the solution (known as the vanishing gradients problem). But the opposite problem is also possible, that the gradients grow, and hence the updates of the weights are larger in each iteration (known as the exploding gradients problem). It has been shown by Glorot and Bengio (2010) that other activation functions than sigmoid behave much better in neuronal networks (see figure 3.7).

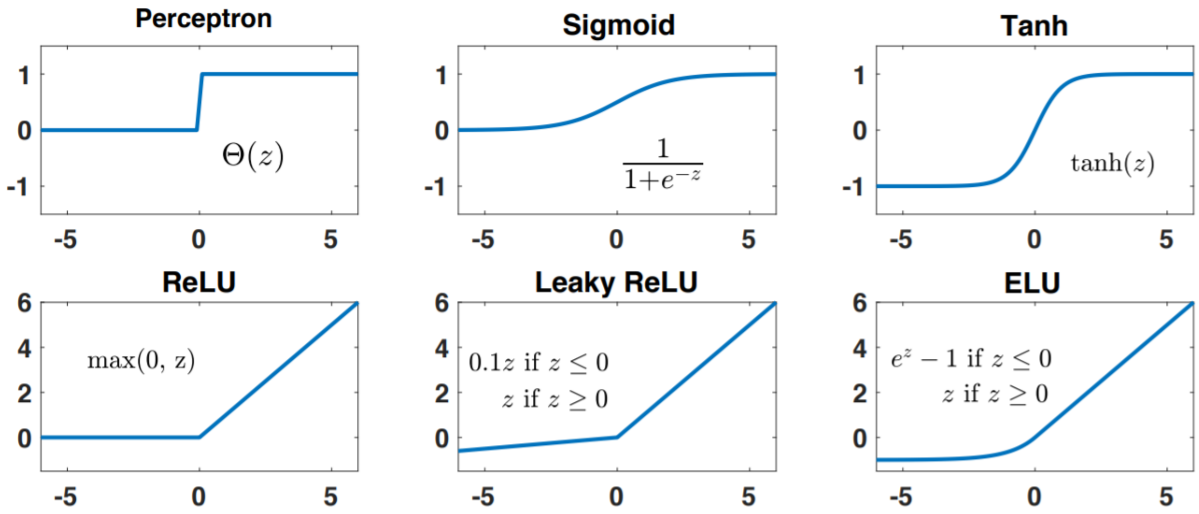

Figure 3.7: Different activation functions and their formulas [9].

3.5.3 Architecture

A neural network consisting of at least three layers, the input layer, the output layer, and the hidden layer in the middle is called a Multilayer Perceptron. The example of figure 3.8 has two hidden layers, the hidden layers consist of six neurons, and there is one single output.

In a feedforward neural network, the output from one layer is the input to the next one, so the information is always sent forward [11]. It is possible to have loops in other models, such as recurrent neural networks, but they have been less influential than feedforward neuronal networks.

We can also distinguish between hard and soft classifiers. A binary model, which only needs one neuron in the output layer, would be an example of a hard classifier because it outputs the class of the input directly. A classifier that outputs the probability of the class is a soft classifier. An example of this is the sigmoid function [6].

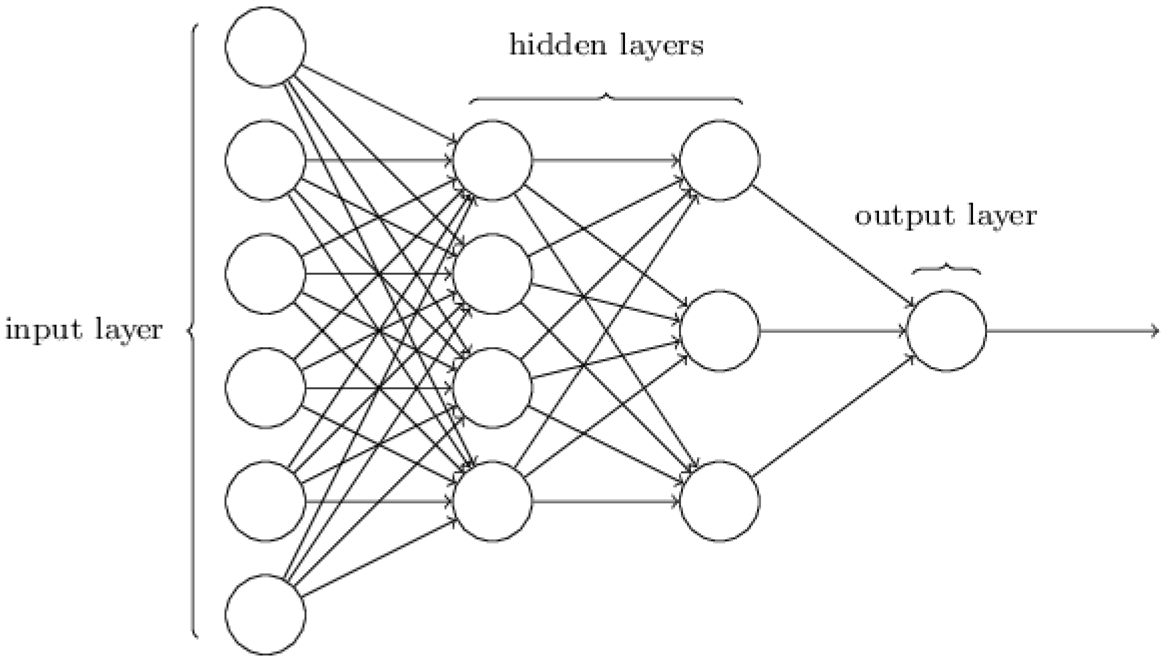

Figure 3.8: Illustration of a feedforward neural network with two hidden layers [11].

3.5.4 Forward and Backpropagation algorithm

Until this point, we have been looking at the forward propagation. It consists of calculating the weighted sum of the input features to each neuron in the hidden layer, then pass it through the activation function, and calculate the weighted sum of the activations in the hidden layer to each neuron in the output layer, to finally calculate the output. [6]

In the backpropagation algorithm, we study how altering the weights and biases affect the cost. The following calculations are based on two assumptions. The first one being that the cost function can be written as the average of the cost functions for all the individual training examples. The second one is that we assume it can be written as a function of the output from the neural network. These are both true for the cost functions used in this article.

Let L denote the output layer, l the index of an arbitrary layer, δ^l be the error in layer l, C be the cost function and b denote bias and w denote weights, σ denote a arbitrary activation function.

$$\delta^L = \frac{\partial C}{\partial a_j^L} \sigma' \left(z_j^L\right) = \left(a_j^L - y\right) \sigma' \left(z_j^L\right) \tag{3.35}$$

$\frac{\partial C}{\partial a_j^L}$ describes how fast the cost changes as a function of the j^{th} activation function. While $\sigma' \left(z_j^L\right)$ describes the change in the activation function at the output layer. The product of these terms expresses the error in the output layer.

$$\delta^l = \left(\left(w^L\right)^T \cdot \delta^L\right) \sigma' \left(z_j^l\right) \tag{3.36}$$

$\left(\left(w^L\right)^T \cdot \delta^L\right)$ backpropagates the error in the output layer, and $\sigma' \left(z_j^l\right)$ describes the change in the activation between these to layers. The product of the two terms describes the error in the hidden layer. This can be written more generally with $l+1$ instead of L.

$$\frac{\partial C}{\partial b_{jk}^l} = \delta^l \tag{3.37}$$

Equation 3.37 describes that the rate of change of the cost as a function of the bias is the error of a layer.

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta^l \tag{3.38}$$

The rate of changes to the cost with respect to the weights of a certain layer, l, is the product of the activation in layer $l-1$ and the error in the layer l. The backpropagation algorithm was introduced in the late 80's and has then been a central part of the learning process of neural networks. Nielsen (2015) [11].

3.6 Structure and implementations

The structure of both the logistic regression and a general neural network is inspired by the structures found in the book by Raschka (2017). We studied several activation functions, varied the penalties for ridge and the LASSO, and tested different initialisation of the weights and bias, $w \in (-1, 1)$ and $b = 1$. This is similar to the logistic regression in scikit-learn, making it simpler to compare. We tested two different optimizers, steepest descent and stochastic gradient descent with mini-batches (also known as SGD). Since the performance of the model was not good with the steepest descent, we decided to use only the SGD that adapts better to our large amount of data, preserving a better structure on the program.

3.6.1 Implementation of Steepest gradient descent

The Steepest descent method was implemented as follows.

Code Listing 3.1: Python pseudo-code of Steepest Descent

```
# 1) Generate a 2-class classification problem (X, y)

# 2) Initialise our weight matrix, w, so it has the same number of
# columns as our input features.
w = 0.1*np.random.randn(X.shape[1])
# Initialise the bias, b.
b = 1

# 3) Loop over the desired number of epochs
for epoch in n_epochs:

    # Take the dot product between ‘X‘ and ‘W‘, add the bias B, and then pass this
    # value through the activation function
    preds = activation_function(X.dot(W)+B)

    # Determine our ‘error‘
    error = preds - y

    # Compute the total loss value as the sum of squared loss
    loss = np.sum(error ** 2)

    # Calculate the gradient
    gradient = X.T.dot(error)

    # Update weights
    w = w - eta * gradient
    b = b - eta * error.sum()
```

The implementation of the stochastic gradient descent with mini-batches is similar to the steepest descent, but computes the gradient over a sample, batch, instead of the entire data set. This means that there is one more loop over the batches nested inside the loop over the epochs in step 3 in the pseudo-code for the steepest descent.

Employing a batch size larger than one has two advantages, it reduces the variance in the parameter update leading to more stable convergence, and it allows us to use highly optimised matrix operations in the computation of the cost and gradient. [16]. Bearing in mind that the matrix operations are more efficient when the matrix size is a power of 2, it is recommended to use a batch size which also is a power of 2. Another difference in the implementation for this variant is that we randomise the training sample before applying the method.

3.6.2 Implementation of stochastic gradient descent with minibatches

https://wiseodd.github.io/techblog/2016/06/21/nn-sgd/

Code Listing 3.2: Python pseudo-code of Stochastic Gradient Descent with Minibatches

```
def sgd(model, X_train, y_train, minibatch_size):
    for epoch in range(n_epochs):
        # Randomize data point
        self.random.shuffle(indices)

        for i in range(0, X_train.shape[0], minibatch_size):
            # Get pair of (X, y) of the current minibatch
            batch_idx = indices[idx:idx + self.batch_size]
            batchX = X_train[batch_idx ,:]
            batchY = y_train[batch_idx]

            model = sgd_step(model, X_train_mini, y_train_mini)

    return model
```

The code reproduces the results accomplished by Mehta et al. (2018)[9], as well as the obtained with Python libraries such as scikit-learn. The comparisons are reproducible with the Jupyter notebooks saved in the GitHub repository.

3.6.3 Implementation of Neuronal Networks

The perceptron model can be summarised by the following steps[13]:

1. Initialise the weights, w, to zero or small random numbers.
2. For each training sample $x^{(j)}$:
   (a) Compute the output value $\hat{y}$.
   (b) Update the weights.

Typically weights are initialized with small values distributed around zero, drawn from a uniform or normal distribution. Setting all weights to zero means all neurons give the same output, making the network useless.

Adding a bias value to the weighted sum of inputs allows the neural network to represent a greater range of values. Without it, any input with the value 0 will be mapped to zero (before being passed through the activation). The bias weights ˆb are often initialized to zero, but a small value like 0.01 ensures all neurons have some output which can be backpropagated in the first training cycle.

Code Listing 3.3: Python pseudo-code of neuronal networks

```
    def _backprop(self, y_train, X_train, A_hidden, Z_hidden, A_out, Z_out, batch_idx):
        """ Backpropagation algorithmn for MLP with  one hidden layer

        X_train : array, shape = [n_samples, n_features]
            Input layer with original features.
        y_train : array, shape = [n_samples]
```

```
            Target class labels or data we want to fit.
        Z_hidden : (array-like)
          Signal into the hidden layer.
        A_hidden : (array-like)
          The activated signal into the hidden layer.
        Z_out : (array-like)
            Signal into the output layer.
        A_out : (array-like)
            Activated signal function.
        batch_idx : int
            The index where you iterate from.
        """

        delta_a_out = A_out - y_train[batch_idx].reshape(self.batch_size, 1)

        if regression network:
            act_derivative_out = self.activate(Z_out, ”linear”, deriv = True)
        elif classification network:
            act_derivative_out = self.activate(Z_out, ”sigmoid”, deriv = True)
        else:
            raise ValueError('Invalid␣activation␣function␣{}'.format(self.tpe))
        # Since we are in the regression case with a linear ouput funct.

        delta_out = delta_a_out*act_derivative_out
        grad_w_out = np.dot(A_hidden.T, delta_out)
        grad_b_out = np.sum(delta_out, axis=0)

        self.W_out = self.W_out - self.eta * grad_w_out
        self.b_out = self.b_out - self.eta * grad_b_out

        act_derivative_h = self.activate(Z_hidden, self.activation, deriv=True)
        error_hidden = np.dot(delta_out, self.W_out.T) * act_derivative_h
        grad_w_h = np.dot(X_train[batch_idx].T, error_hidden)
        grad_b_h = np.sum(error_hidden, axis=0)

        self.W_h = self.W_h - self.eta * grad_w_h
        self.b_h = self.b_h - self.eta * grad_b_h

        return None


    def predict(self, X):
        """Predicts outcome of regression or class labels depending
        on the type of network you have initialised.

        Returns:
        ----------
        y_pred : array, shape = [n_samples]
            Regression nn : predicts outcome of regression.
            Classification : predicts class labels.
        """

        Z_hidden, A_hidden, Z_out, A_out = self._forwardprop(X)

        if classification network:
            return np.where(A_out >= 0.5, 1, 0)
        elif regression network:
            return A_out

def _minibatch_sgd(self, X_train, y_train):
        """Performes the stochastic gradient descent with mini-batches for one epoch."""
        n_samples, n_features = np.shape(X_train)

        indices = np.arange(n_samples)

        if self.shuffle:
            self.random.shuffle(indices)

        for idx in range(0, n_samples, self.batch_size):

            batch_idx = indices[idx:idx + self.batch_size]

            # Forwardpropagate
            Z_hidden, A_hidden, Z_out, A_out = self._forwardprop(
                X_train[batch_idx, :]
            )

            # Backpropagate
            self._backprop(
                y_train, X_train, A_hidden, Z_hidden, A_out, Z_out, batch_idx
            )

        return self

    def fit(self, X_train, y_train, X_test=None, y_test=None):
        """ Learn weights from training """

        self.initialize_weights_and_bias(X_train)

        # iterate over training epochs
        for epoch in range(self.epochs):

            # Includes forward + backward prop.
            self._minibatch_sgd( X_train, y_train)

            # Evaluation after each epoch during training
            z_h, a_h, z_out, a_out = self._forwardprop(X_train)

            y_train_pred = self.predict(X_train)
            y_test_pred = self.predict(X_test)

            y_test = y_test.reshape((len(y_test),1))
            y_train = y_train.reshape((len(y_train),1))

            if regression network:
                # Calculate mean squared error
                train_preform = mean_squared_error(y_train, y_train_pred)
                valid_preform = mean_squared_error(y_test, y_test_pred)
                self.eval_['train_preform'].append(train_preform)
                self.eval_['valid_preform'].append(valid_preform)

            elif classification network:
                #Calculate accuracy
                acc_test = np.sum(y_test == y_test_pred)/len(y_test)
                acc_train = np.sum(y_train == y_train_pred)/len(y_train)
                self.eval_['train_preform'].append(acc_train)
                self.eval_['valid_preform'].append(acc_test)

        return self
```

For more details about methods and implementation of regression analysis, see the report and the code of project 2 in the GitHub repository.

Chapter 4

Results

The results discussed in this section are obtained with our own implementation of the models, explained in the previous sections, and are in agreement with the benchmark results from Mehta et al. 2016 [9]. In addition, we compare the results from our own code to the generated with standard python libraries for machine-learning as scikit-learn. For more details, see the notebooks in the GitHub link for this project.

4.1 Linear regression and more advanced regression methods on one dimensional Ising model

Linear regression analysis was applied to the one-dimensional Ising model in order to learn the Hamiltonian 3.1. The approaches considered are the OLS, ridge, and the LASSO. The range of temperatures, T was restricted to $T \in [0.25, 1.75] \cup [2.75, 4.0]$, since we are only interested in the ordered and disordered phases. For this purpose, we assume that an ensemble of random spin configurations was generated from the one-dimensional Ising model, with J=1, and that we are provided the dataset $D = (\{S_j\}_{j=1}^{L}, E_j)$. The meaning of E_j is unknown, and we want the model to learn to predict the values of E_j from the spin configurations $\{S_j\}_{j=1}^{L}$. Considering that we do not know how the data set was originated, it is reasonable to employ a spin model with pairwise interactions between every pair of variables 3.2, but recast as in 3.3.

The performance of the three approaches on the training and test set is illustrated in figure 4.1. The MSE score is shown in the left panel whereas the R-square is shown in the right panel, in both cases as a function of the regularisation parameter, λ, which varies from 10^{-4} to 10^{4}. It is clear that the scores are better for the test set than for the training test, indicating that there is no overfitting. The figure reveals, in the case of ridge regression, that the penalisation does not improve the fit. As a consequence, the scores of the ridge are not better than those for the OLS. This is not the case for the LASSO, which clearly performs better than OLS for low values of λ. The curves for the OLS and ridge are monotonic, but not for the LASSO (even though it is hard to see in the figure). The optimal regularisation parameter for the LASSO in regard to the errors is $= 0.001$ in the table 4.2. When selecting this value of λ, the Ising interaction terms, J, are other than zero only for the nearest-neighbour terms, see figure 4.2. This is the only model that reproduces the original pattern from which the data was generated. Recall that $J_{k,k} = J = 1$ was not defined to be symmetric, since only the $J_{j,j+1}$ were used.

Figure 4.2 exhibits the matrix representation of $J_{j,k}$ for the optimal parameters and the three approaches. A figure with more regularisation parameters can be found in the appendix (6.1). Both OLS and ridge show a symmetric behaviour with values close to -0.5 for $J_{j,j-1}$ and $J_{j,j+1}$ terms; the remaining elements of the diagonal present some noise. The LASSO, on the other hand, results in values of $J_{j,j+1}$ close to 1, and values of $J_{j,j-1}$ close to 0, breaking the symmetry. The LASSO models with $10^{-3} < \lambda < 10^{-1}$ are the only models tested that capture this asymmetry.

Table 4.1 shows that the MSE of the test sample for the LASSO model with the optimal lambda is 3.07, much lower than the MSE for the OLS and ridge. The bias and the variance are also lower for the LASSO. All the schemes present a lower MSE than the sum of the variance and the bias, even though the difference is small ($< 10^{-12}$).

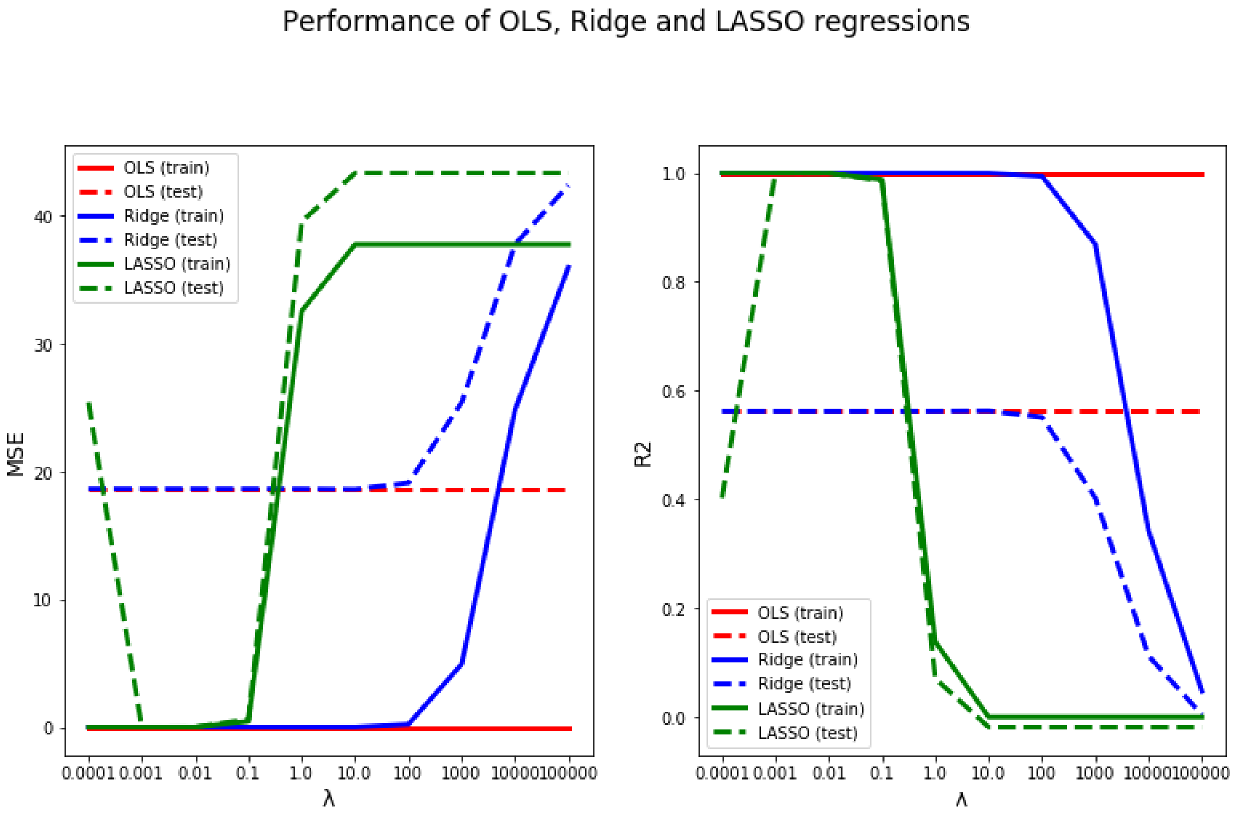

Figure 4.1: Performance metrics with Trainsize = 400 and testsize = 200.

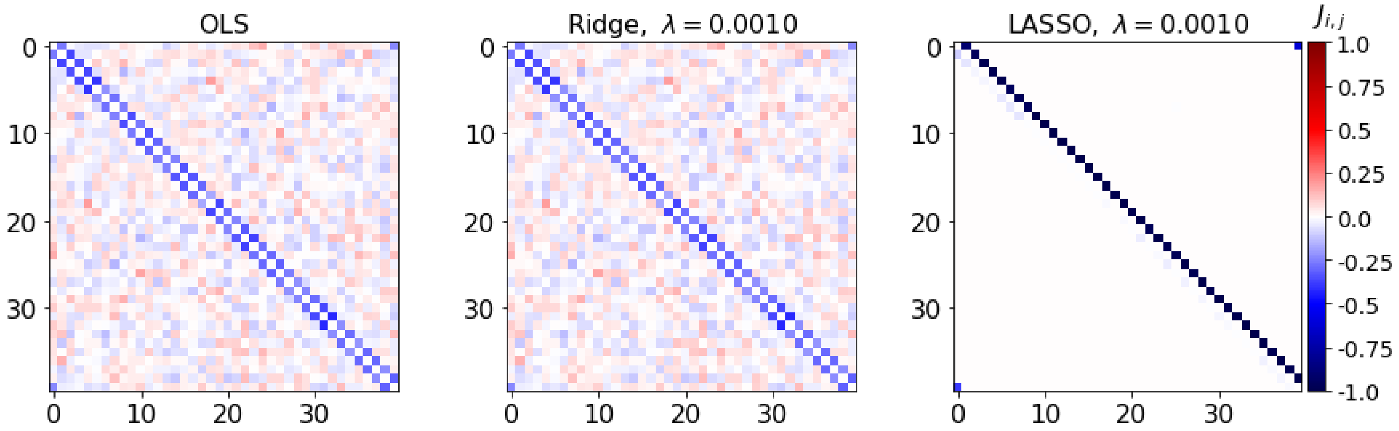

Figure 4.2: The coefficients for the best results of each of the three models.

	Variance	Bias^2	MSE
Ridge	4.0852	21.9330	25.9931
Lasso	1.3374	1.7331	3.0705
OLS	4.0852	21.9330	26.0183

Table 4.1: Bias variance tradeoff.

The regression analysis was also performed employing our own code for neuronal networks, utilising the sigmoid as activation function for the OLS approach. The performance of two of two different MLP models with sigmoid as activation function and different learning rates ($\eta = 0.0001$ and $\eta = 0.0010$ is shown in figure 4.3 as a function of the number of epochs. In both cases the MSE of the training set is lower than the MSE of the test set, indicating some possible overfitting, even though the difference is small. We can observe that the curves of the different models intersect around 20 epochs, due to the fact that the MSE for the model with the smallest eta decreases much faster than the other. This proves the importance of choosing a high enough number of iteration, in order to select the best hyperparameters. After a certain number of iterations, the error curves flatten, meaning that it is computationally inefficient to loop more times over the data.

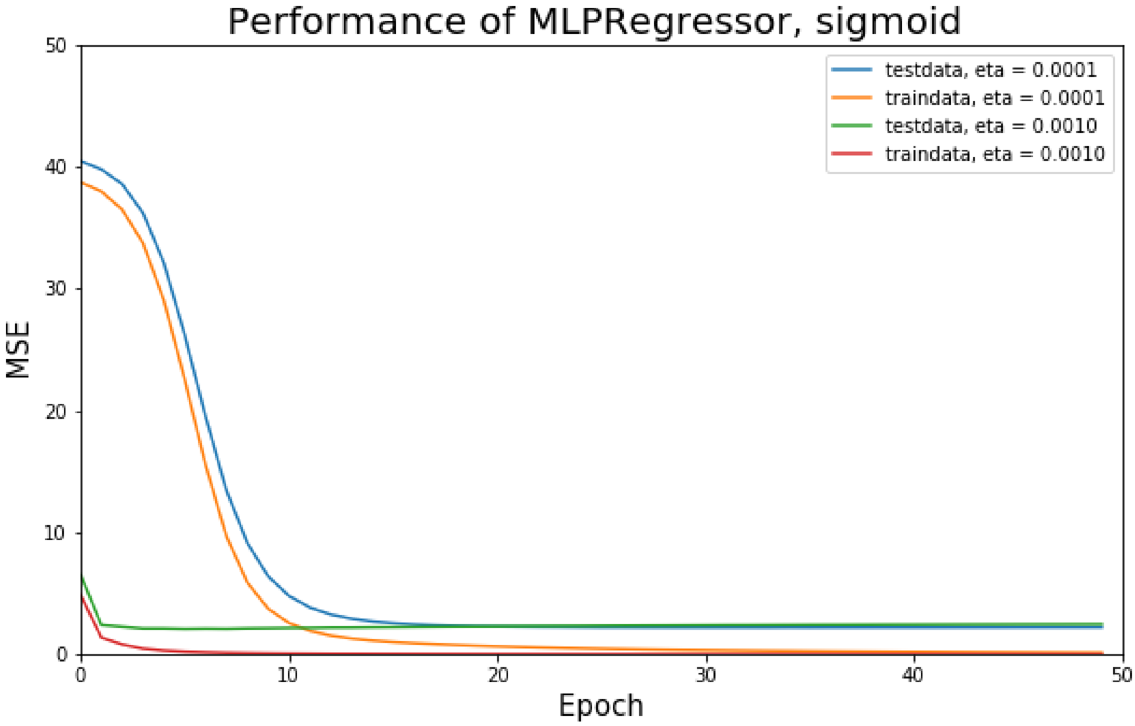

Figure 4.3: Illustrates the training of the MLPRegressor as a function of epochs. This is run for 50 epochs with a batch size of 10 samples.

Table 4.4 exhibits the MSE calculated with the test data of the OLS approach for 1, 10, 50 and 100 epochs, batch sizes 1, 10 and 50, and $\eta = 10^{-2}, 10^{-3}, 10^{-4}$. We run the same epoch batch combination for the learning rates $\eta = 0.1$ $\eta = 1.0$, but this only resulted in nan values. This occurs as a result of exploding output weights. From 4.3 we can see that the training MSE for both learning rate converge to the same value, this is also the case for the training data.

It is clear that the MSE decreases more rapidly for $\eta = 10^{-4}$ than for $\eta = 10^{-3}$, however, the latter scores better. Restricting the models to this learning rate, there is no evident improvement for more than 10 epochs. Besides, it is not clear which batch size gives better results.

Epoch	Batch size	$\eta = 0.0001$	$\eta = 0.001$	$\eta = 0.01$
1	1	39.65	7.59	4.25
	10	40.12	6.52	inf
	50	39.9	9.69	$5 \cdot 10^{230}$
10	1	25.23	2.62	3.06
	10	24.25	2.55	nan
	50	27.67	2.56	nan
50	1	7.01	2.56	3.37
	10	7.20	2.48	nan
	50	6.93	2.60	nan
100	1	4.72	2.72	3.66
	10	4.60	2.77	nan
	50	4.60	2.65	nan

Table 4.2: This table contains the MSE values averaged over all epochs.

4.2 Classification - logistic regression vs multiple perceptron model, neural network

In this section, we determine the phases, ordered and disordered, of the two-dimensional Ising model, given the spin configuration. We apply logistic regression and neuronal networks on a fixed lattice $40x40$ of spins, assuming that $J = 1$. Temperatures that lead to a critical phase are not used in this analysis since the finite lattice does not show a clear sign of a phase transition.

The performance of the test data is evaluated with the accuracy score. The optimal parameters are found using the stochastic gradient descent method with a fixed learning rate when looping over the batches. The steepest descent method was also implemented but discarded for computational reasons. As discussed before, the stochastic gradient descent is a better option when the data sets are big, since it does not iterate over the entire data set, but only over random batches. The batch size used is 10.

The accuracy for the ridge and LASSO regression is provided in terms of two hyperparameter, the regularization parameter and the learning rate η. Figures 4.6, 4.7, exhibit the accuracy in a heat map for $\eta = 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$, $\lambda = 10^{1}, 10^{0}, 10^{-1}, ..., 10^{-4}$ and 50 epochs. Different number of epochs (iterations) were tested, and it is clear that the accuracy is incremented rapidly with the number of iterations until certain number of epochs is reached, then it becomes stable. This relationship is shown in figure 4.4. Here, the cost function in is plotted for up to 100 epochs; it decreases rapidly until 40 epochs, and at 60 it becomes stable.

The figures for only one epoch can be found in the appendix, and they are presented as an evidence of that low number of epochs leads to lower accuracy. We have also saved the figures made with different activation functions and number of epochs, but the discussion of these is reserved for a future article since they have to be analysed carefully with more elements than the used in this study.

The accuracy of logistic regression for the LASSO with a regularisation parameter and a learning rate both equal to 10^{-4} for the training and the test data is represented in figure 4.5. The first thing we want to point out is that the accuracy is very sensitive to the number of epochs, varying from approximately 0.45 to 0.70, but we cannot distinguish any particular pattern. This means that the number of correctly classified phases depends strongly on the number of epochs chosen. Looking at figures 4.4 and 4.5, we have chosen 50 epochs, since the cost is low and the accuracy is relatively high for this number.

The accuracy for different ridge models using sigmoid as activation function is presented in figure 4.6. It is evident that the best results are accomplished for $\eta = 0.01$, and penalisation parameter $\lambda = 10^{-2}, 10^{-3}, 10^{-4}$, in fact, even though there are small differences in accuracy, the highest accuracy of 0.71 corresponds to the combination of $\lambda = 0.001$, $\eta = 0.001$ and 10 epochs. For one epoch, an accuracy of 0.70 is reached with $\eta = 10^{-4}$ and $\lambda = 0.1$, but in general, the accuracy is lower for different combinations of η and λ. The figure indicates that the performance is improved with respect to OLS, which is a particular case with no regularisation and presents and accuracy lower than 0.47 (see table 4.3).

The highest accuracy using LASSO and sigmoid (also 50 epochs) is 0.69 for the lowest values of η and λ

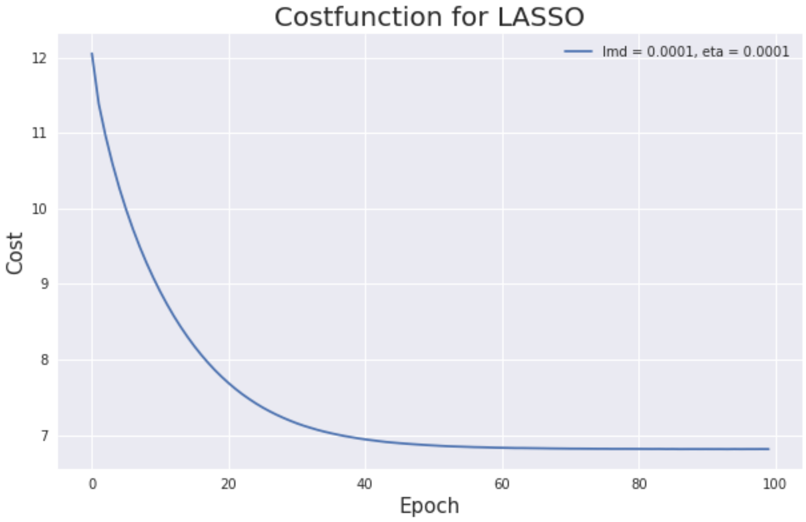

Figure 4.4: Costfunction lasso 100 epoch using sigmoid activation function and logistic regression

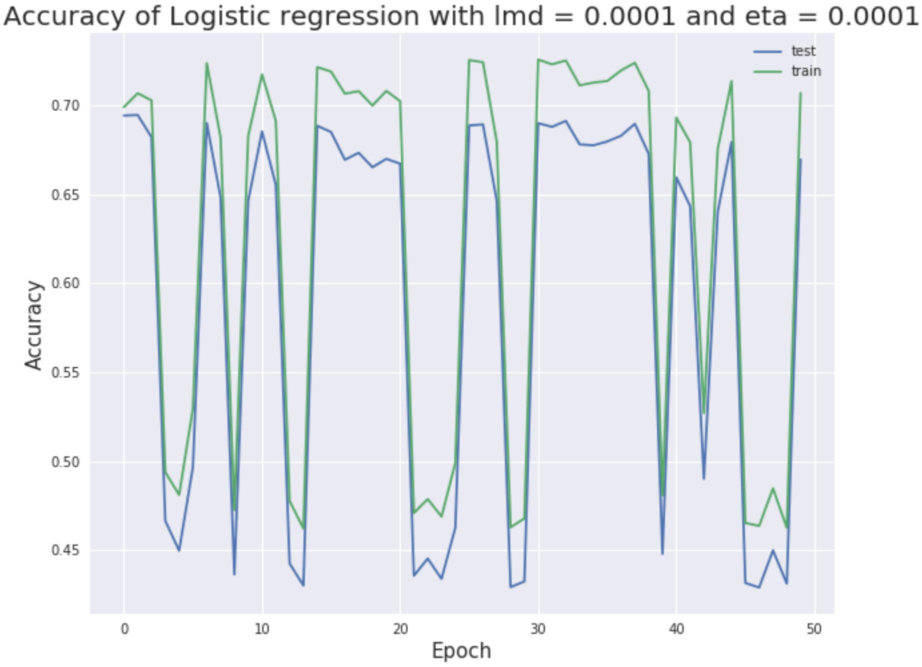

Figure 4.5: Accuracy logistic regression with a lasso penalty of 0.0001 and η = 0.0001, batch size = 10 as a function of epochs.

	Accuracy OLS
$\eta = 0.0001$	0.447
$\eta = 0.001$	0.464
$\eta = 0.01$	0.469
$\eta = 0.1$	0.469

Table 4.3: Logistic regression accuracy for different learning rates, without a penalty.

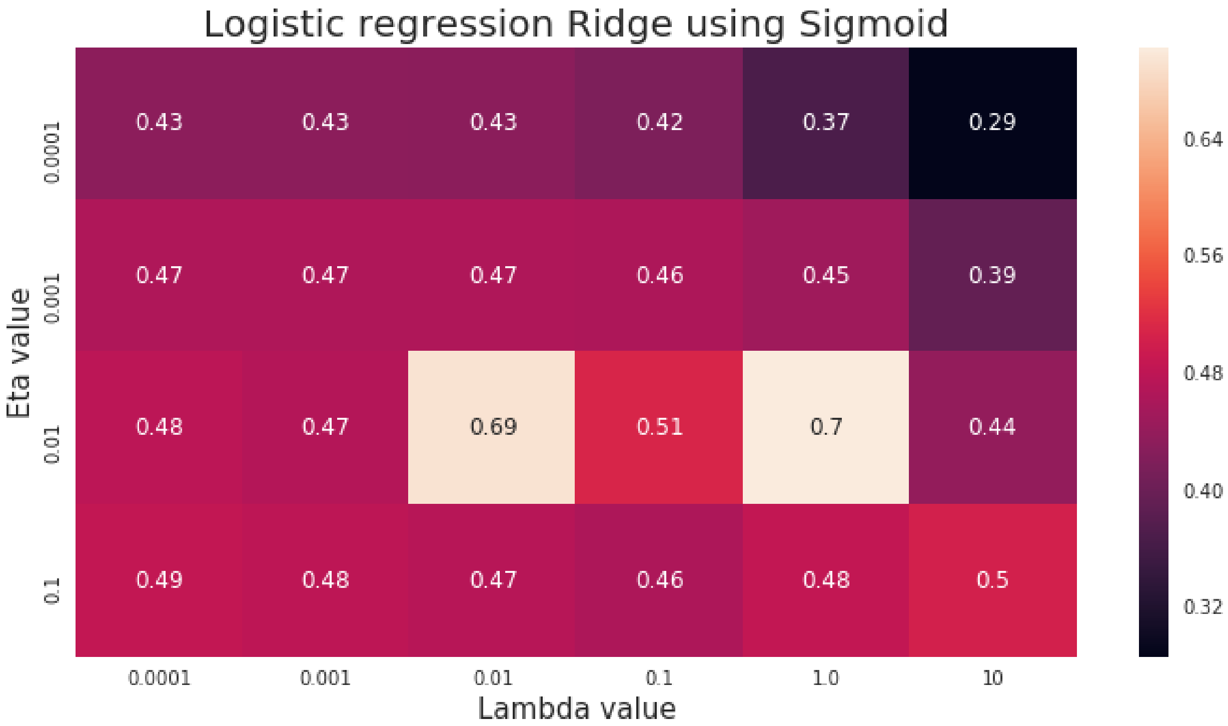

Figure 4.6: Heatmap showing the accuracy of logistic regression with a ridge penalty. Using sigmoid as activation function with 50 epochs and a batchsize of 10 samples.

tested (see figure 4.7). This means that the ridge performs better than the LASSO when using the sigmoid function and the same number of epochs. However, LASSO reaches an accuracy of 0.71 using 10 epochs (see appendix). Other functions as ELU and leaky ReLU were also tested for the LASSO approach, but the performance is not improved. This may be explained by the Runtime warnings we got. These arise from overflow and invalid numbers.

A neuronal network with one hidden layer was also implemented for the classification problem with no regularisation, i.e., the OLS.

As we can see in figure 4.8, the accuracy of the MLP model without regularisation and with $\eta = 0.0001$ using the sigmoid function, increases with the number of epochs, especially from 1 to 10, after that, it continues increasing but the curve is less steep. The test error is higher than the training error, pointing out that there is some overfitting, but the difference is small and the accuracy is high for both data sets when iterating sufficient times.

The accuracy is presented in table 4.4. We used different numbers of epochs, $epochs = 1, 10, 50, 100$, three different batch sizes (1, 10, 50), and learning rates $\eta = 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$. In general, the performance increases with the learning rate, especially from $\eta = 10^{-4}$ to $\eta = 10^{-3}$. The lowest accuracy is 0.73 and the largest is $0.\overline{99}$. It is also evident that the accuracy improves when the number of iterations increases. The batch size seems to be more relevant when both the number of epochs and the learning rate are low.

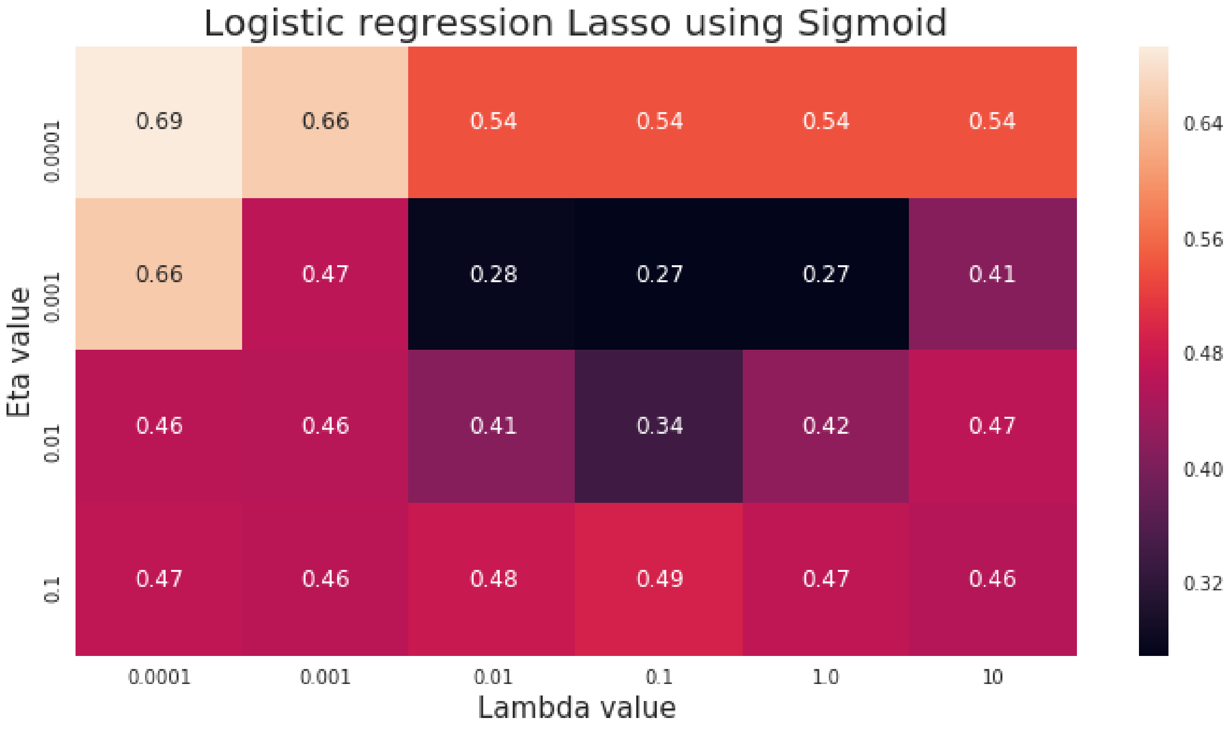

Figure 4.7: Heatmap showing the accuracy of logistic regression with a lasso penalty. Using sigmoid as activation function with 50 epochs and a batch size of 10 samples.

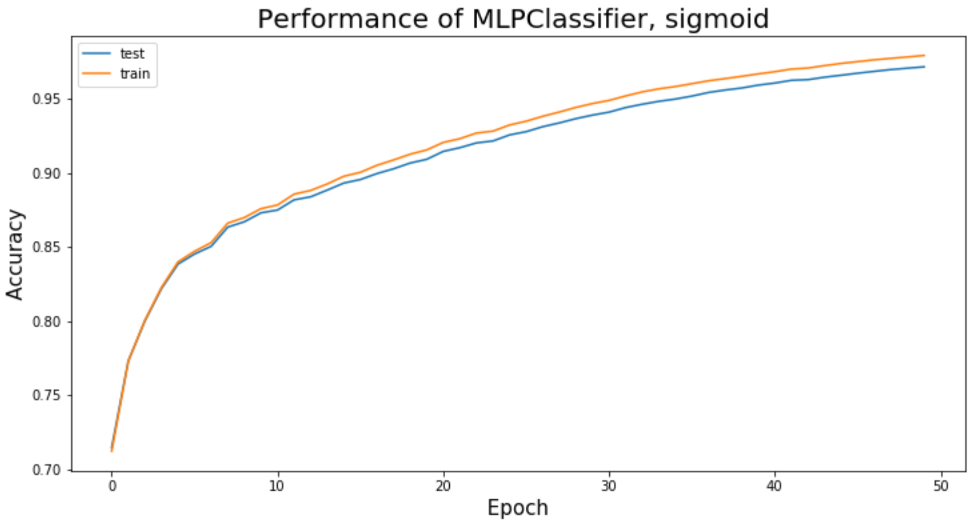

Figure 4.8: Illustrates the training of the MLPClassifier as a function of epochs. This is run for 50 epochs with a batch size of 10 samples.

Epoch	**Batch size**	$\eta = 0.0001$	$\eta = 0.001$	$\eta = 0.01$	$\eta = 0.1$
1	1	0.7301	0.8833	0.9943	0.9989
	10	0.7295	0.8625	0.9949	0.9996
	50	0.7634	0.8760	0.9925	0.9998
10	1	0.8410	0.9595	0.9983	0.9995
	10	0.8400	0.9583	0.998	0.9999
	50	0.8409	0.9593	0.9979	0.9998
50	1	0.9154	0.9886	0.9989	0.9998
	10	0.9186	0.9896	0.9992	0.9999
	50	0.9130	0.9879	0.9939	0.9999
100	1	0.9486	0.9942	0.9994	0.9998
	10	0.9525	0.9940	0.9994	0.9999
	50	0.9510	0.9947	0.9995	0.9999

Table 4.4: Displays the accuracy of the performance of the multilayer perceptron for classification.

Chapter 5

Conclusions

We compared different regression and classification techniques, developing our own code for the OLS, ridge, the LASSO, and logistic regression, as well as the neuronal networks with one hidden layer. At the same time, we experimented with different gradient descent methods, selecting the stochastic gradient descent with minibatches for presenting the most interesting results. Different activation functions were also explored (sigmoid, ELU, and leaky ReLU), finding that the one that adjusts best to our data is the sigmoid. In a future work, we will dive deeper into the topic of activation functions and repeat the analysis with different options. The analysis is presented in terms of some selected hyperparameters, which are the regularisation parameter, the learning rate, the batch size, and the number of epochs. Different values for these will also be studied in the next article, in particular, we wish to investigate why the number of epochs has such a strong effect on the performance.

Good models were found for predicting the Hamiltonian of the one-dimensional Ising model and for classifying the spin configurations of the two-dimensional Ising model. In the regression analysis, the LASSO is the only approach that assigns values close to 1 to the elements $J_{j,j+1}$, and zero to the rest, for $\lambda = 0.001$, and and $MSE = 3.07$. However, neuronal networks improves the learning reaching an $MSE = 2.48$ for 50 epochs, a batch size equal to 10, and $\eta = 0.001$.

Regarding the classification problem, the best results are achieved with the sigmoid function. The LASSO reaches an accuracy of 0.71, in the best case, whereas the ridge gives an accuracy of 0.70. Again, the neuronal network performs better, with an accuracy of $0.\overline{99}$, for $\eta = 0.1$, a high number of epochs and a batch size of more than 10 data points.

In the future, we would like to add penalties to the neuronal networks and experiment with different numbers of hidden layers, varying the number of nodes inside these, and employing other activation functions. It would also be interesting to run the code for batch sizes power of two, and check whether the computation speed improves. Moreover, we would like to compare our results with the outcomes of other libraries such as keras and TensorFlow, as we did with scikit-learn. We underline that the results accomplished are consistent with earlier results presented by Mehta et al. (2018)[9].

Bibliography

[1] A. C. Faul. *A Concise Introduction to Numerical Analysis.* Boca Ranton, Florida: Chapman and Hall/CRC, 2016.

[2] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics.* Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: `http://proceedings.mlr.press/v9/glorot10a.html`.

[3] Jonathan S. Golan. "Moore–Penrose Pseudoinverses". In: *The Linear Algebra a Beginning Graduate Student Ought to Know.* Dordrecht: Springer Netherlands, 2012, pp. 441–452. ISBN: 978-94-007-2636-9. DOI: `10.1007/978-94-007-2636-9_19`. URL: `https://doi.org/10.1007/978-94-007-2636-9_19`.

[4] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[5] Morten Hjort-Jensen. *Lecture notes Data Analysis and Machine Learning: Getting started, our first data and Machine Learning encounters.* `https://compphysics.github.io/MachineLearning/doc/pub/How2ReadData/html/How2ReadData.html`. Accessed: 2018-11-08.

[6] Morten Hjort-Jensen. *Lecture notes Data Analysis and Machine Learning: Neural networks, from the simple perceptron to deep learning and convolutional networks.* `https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/pdf/NeuralNet-minted.pdf`. Accessed: 2018-11-08.

[7] *Lecture notes logistic regression Data Analysis and Machine Learning: Linear Regression and more Advanced Regression Analysis.* `https://compphysics.github.io/MachineLearning/doc/pub/LogReg/html/LogReg.html`. Accessed: 2018-11-11.

[8] *Lecture notes logistic regression DData Analysis and Machine Learning: Logistic Regression.* `https://compphysics.github.io/MachineLearning/doc/pub/Regression/html/Regression.html`. Accessed: 2018-11-11.

[9] P. Mehta et al. "A high-bias, low-variance introduction to Machine Learning for physicists". In: *ArXiv e-prints* (Mar. 2018). arXiv: `1803.08823 [physics.comp-ph]`.

[10] Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller, eds. *Neural Networks: Tricks of the Trade - Second Edition.* Vol. 7700. Lecture Notes in Computer Science. Springer, 2012. ISBN: 978-3-642-35288-1. DOI: `10.1007/978-3-642-35289-8`. URL: `https://doi.org/10.1007/978-3-642-35289-8`.

[11] Micheal Nielsen. *Neural Networks and Deep Learning.* Determintation press, 2015.

[12] *pyimagesearch Gradient Descent.* `https://www.pyimagesearch.com/2016/10/10/gradient-descent-with-python/`. Accessed: 2018-11-06.

[13] Sebastian Raschka. *Python Machine Learning.* Packt Publishing, 2017. ISBN: 1787125939, 9781787125933.

[14] *Stack Overflow blog The Incredible Growth of Python.* `https://stackoverflow.blog/2017/09/06/incredible-growth-python/`. Accessed: 2018-11-11.

[15] *Stack Overflow Developer Survey Results 2018.* `https://insights.stackoverflow.com/survey/2018`. Accessed: 2018-11-11.

[16] *Standford tutorial Optimization of SGD.* `http://ufldl.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent/`. Accessed: 2018-11-08.

[17] *Wikipedia article Ising model.* `https://en.wikipedia.org/wiki/Ising_model`. Accessed: 2018-10-20.

Chapter 6

Appendix

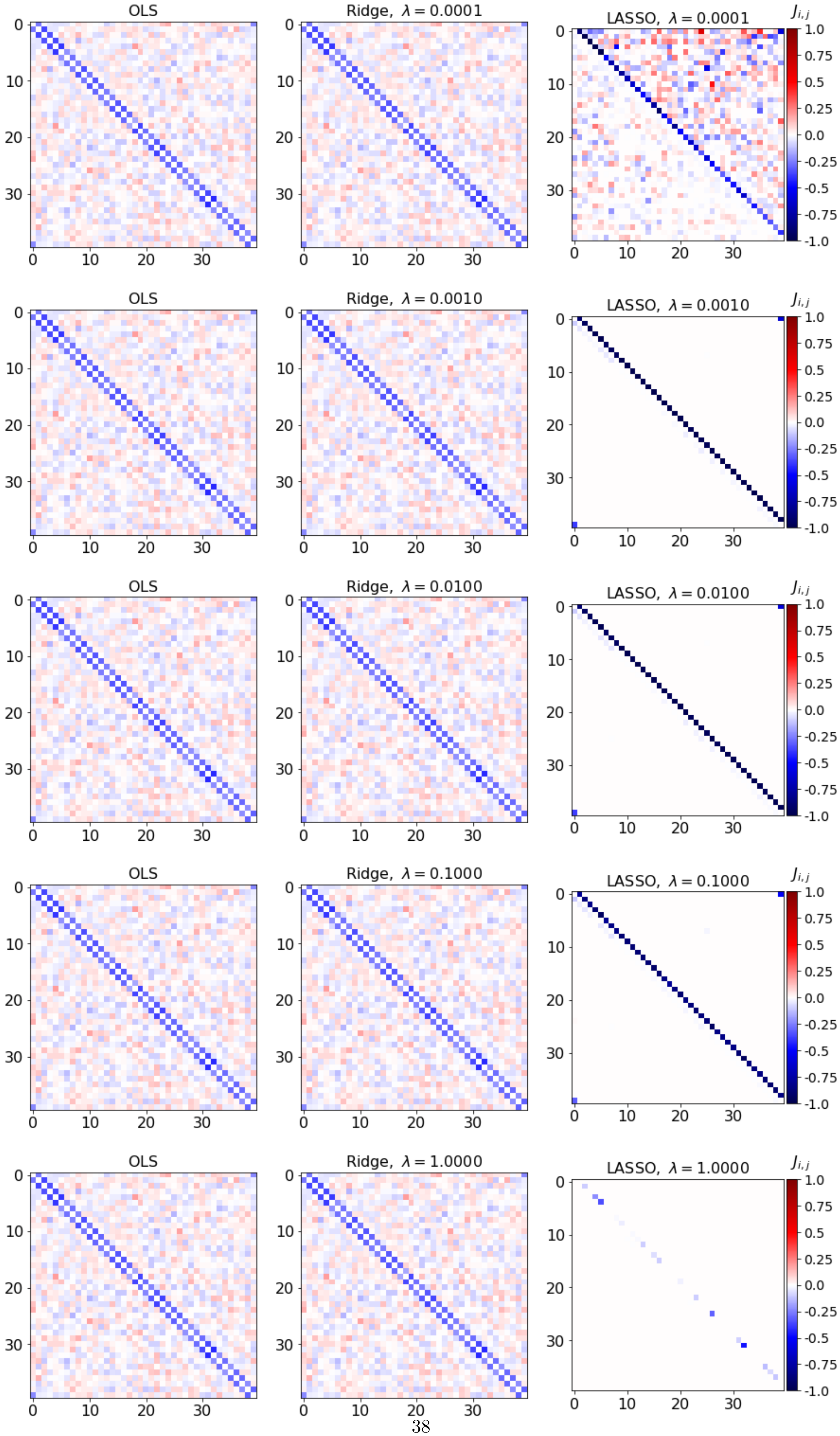

Figure 6.1: This plot shows the coefficient J for all the values of lambda based on the one dimentional data.

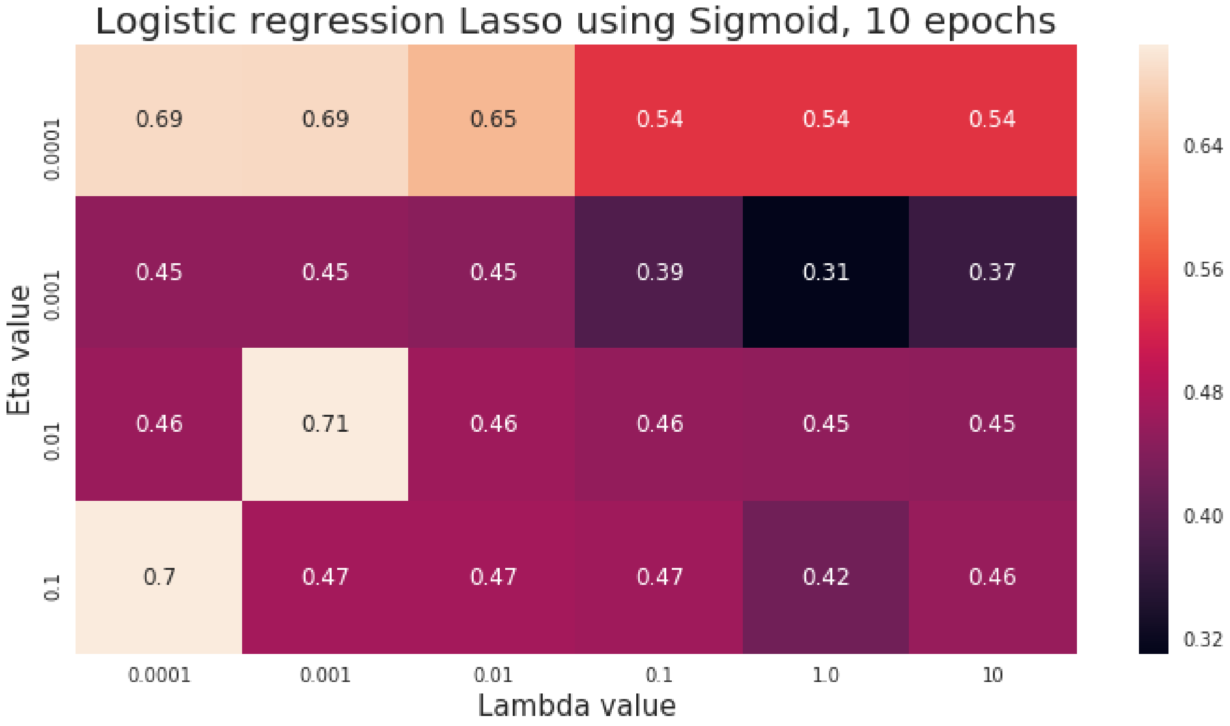

Figure 6.2: Heatmap showing the accuracy of logistic regression with a lasso penalty. Using sigmoid as activation function with 10 epochs and a batch size of 10 samples.

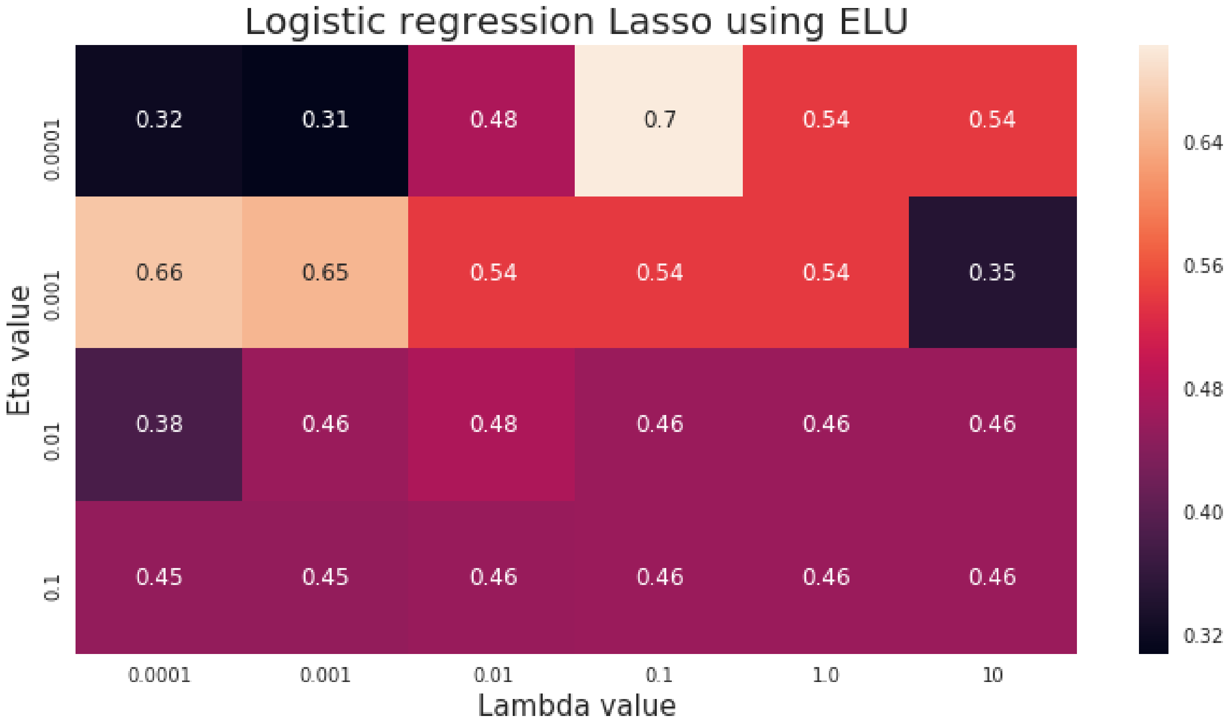

Figure 6.3: Heatmap showing the accuracy of logistic regression with a lasso penalty. Using exponential linear unit, elu as activation function with 1 epochs and a batch size of 10 samples.

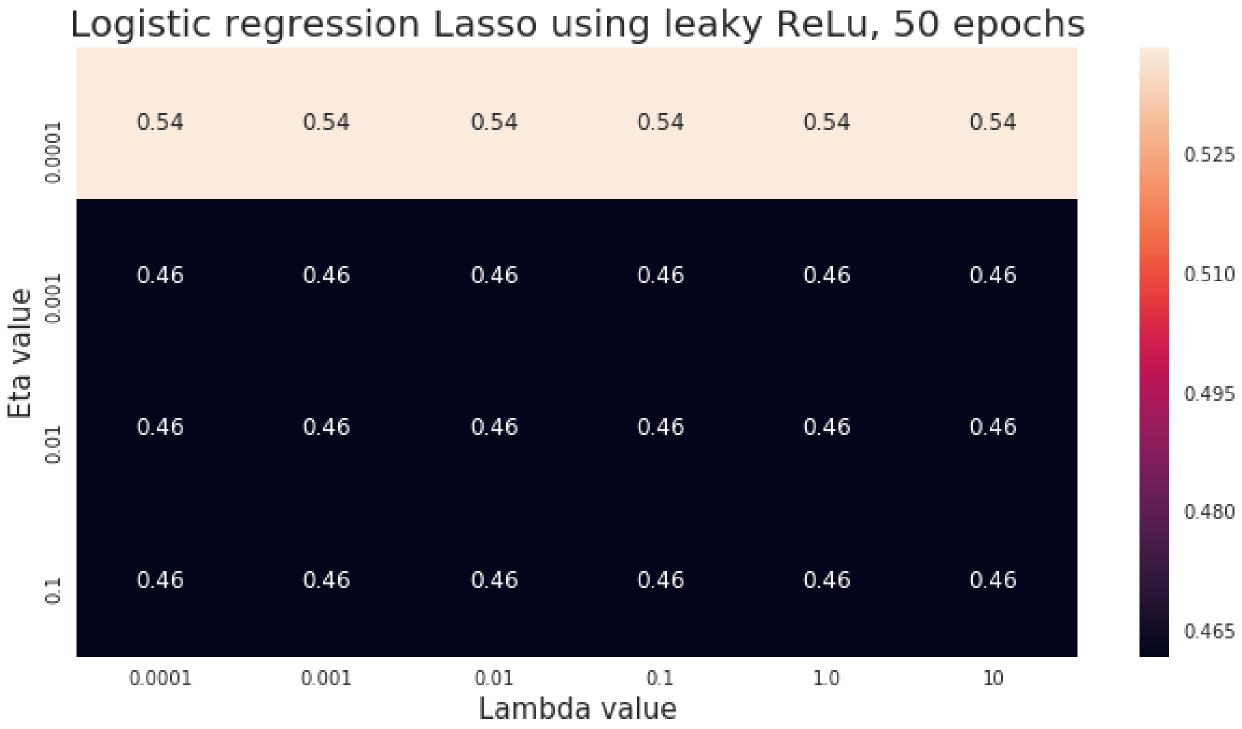

Figure 6.4: Heatmap showing the accuracy of logistic regression with a lasso penalty. Using leaky rectified linear unit, LReLu as activation function with 50 epochs and a batch size of 10 samples.

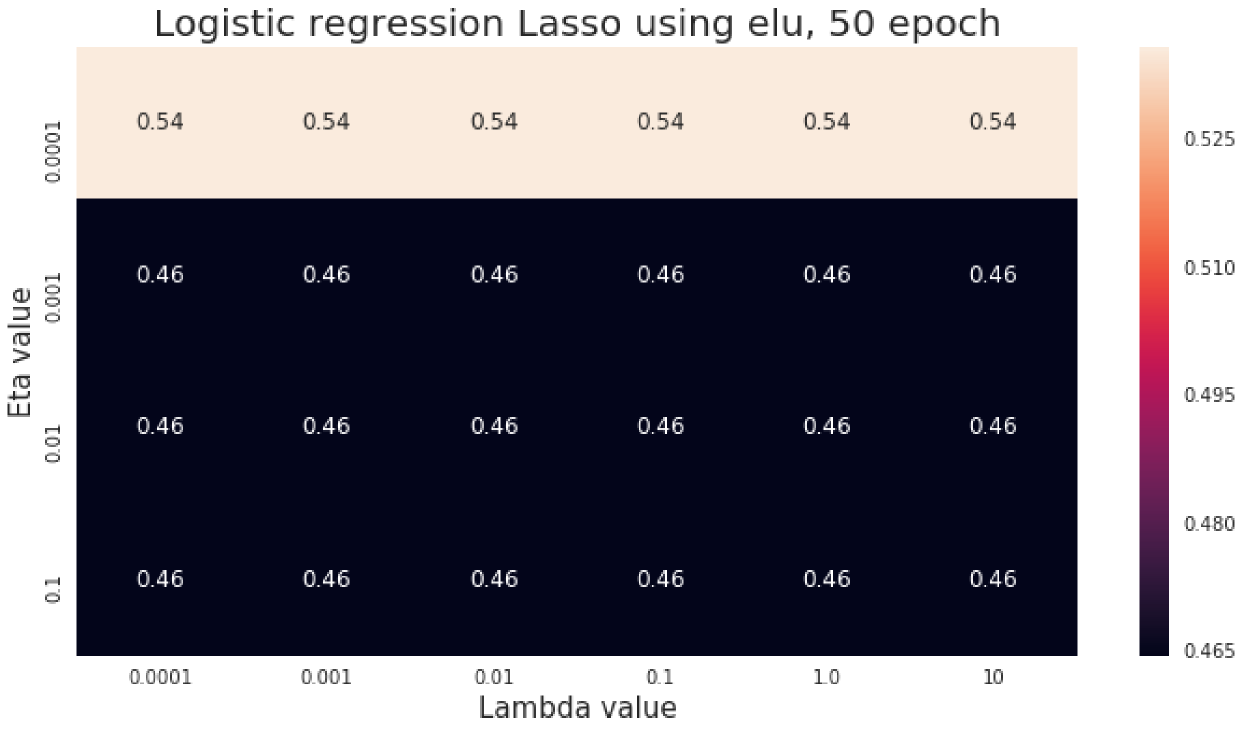

Figure 6.5: Heatmap showing the accuracy of logistic regression with a lasso penalty. Using exponential linear unit, elu as activation function with 50 epochs and a batch size of 10 samples.

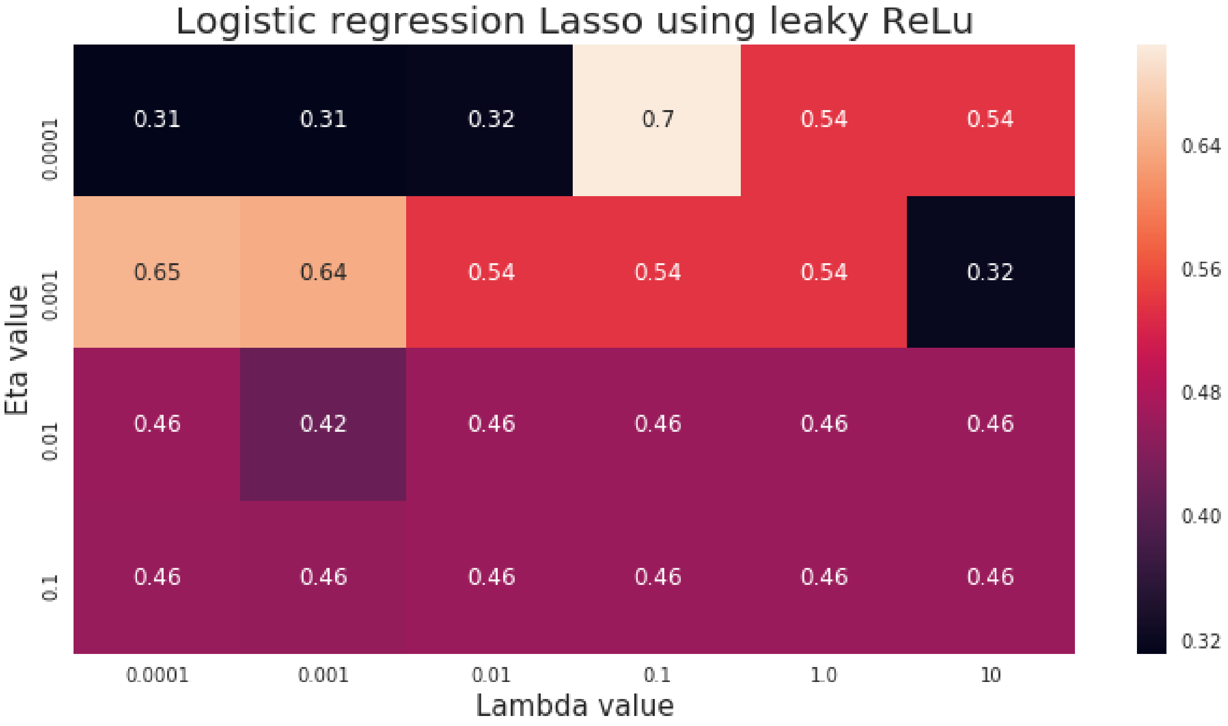

Figure 6.6: Heatmap showing the accuracy of logistic regression with a lasso penalty. Using leaky rectified linear unit, LReLu as activation function with 1 epochs and a batch size of 10 samples.

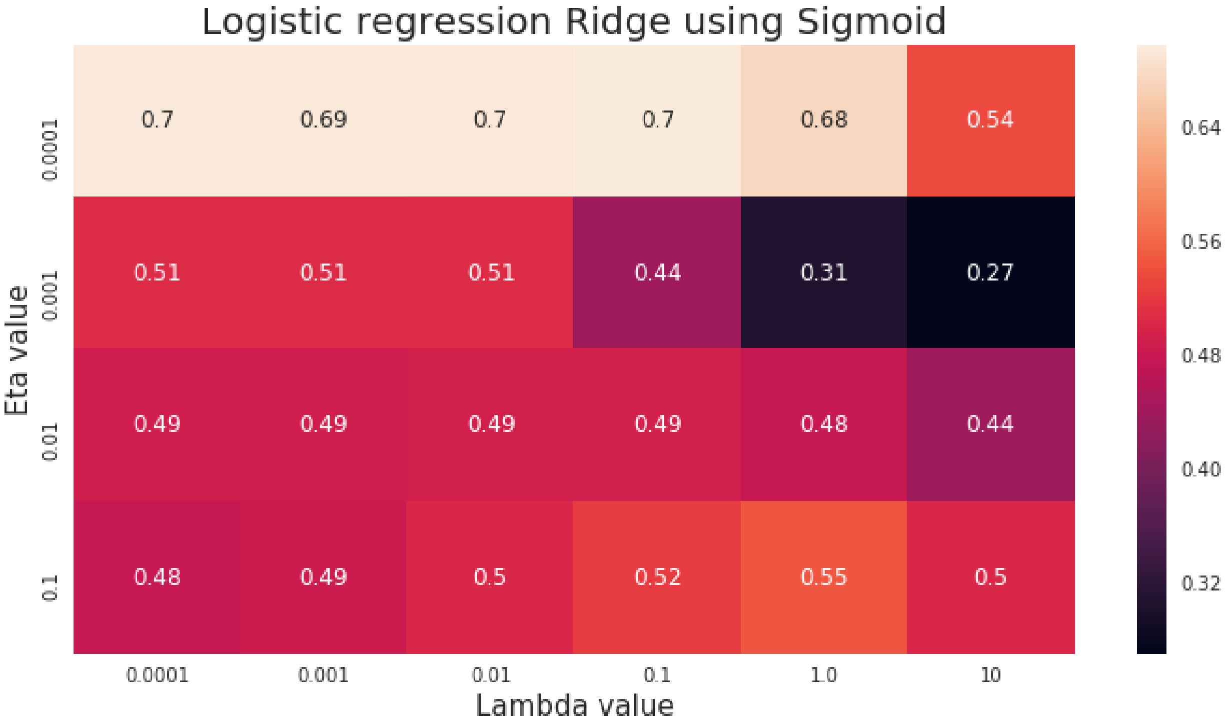

Figure 6.7: Heatmap showing the accuracy of logistic regression with a ridge penalty. Using sigmoid as activation function with 1 epochs and a batch size of 10 samples.

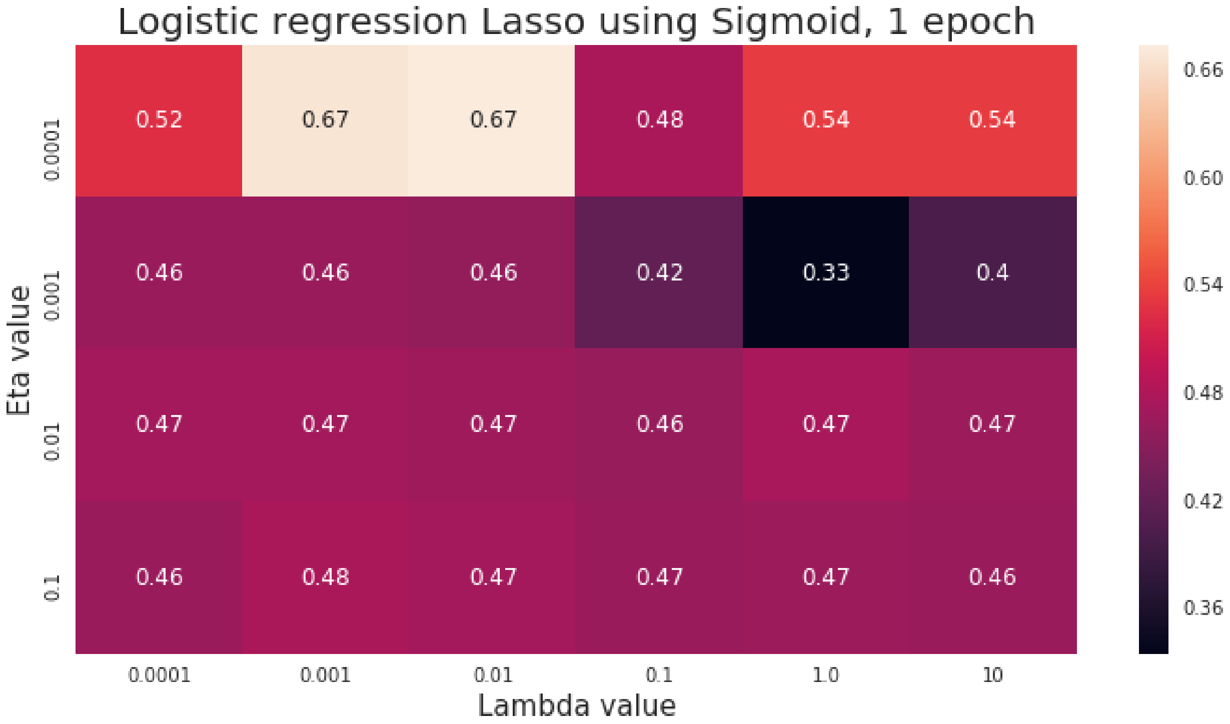

Figure 6.8: Heatmap showing the accuracy of logistic regression with a lasso penalty. Using sigmoid as activation function with 1 epochs and a batch size of 10 samples.